

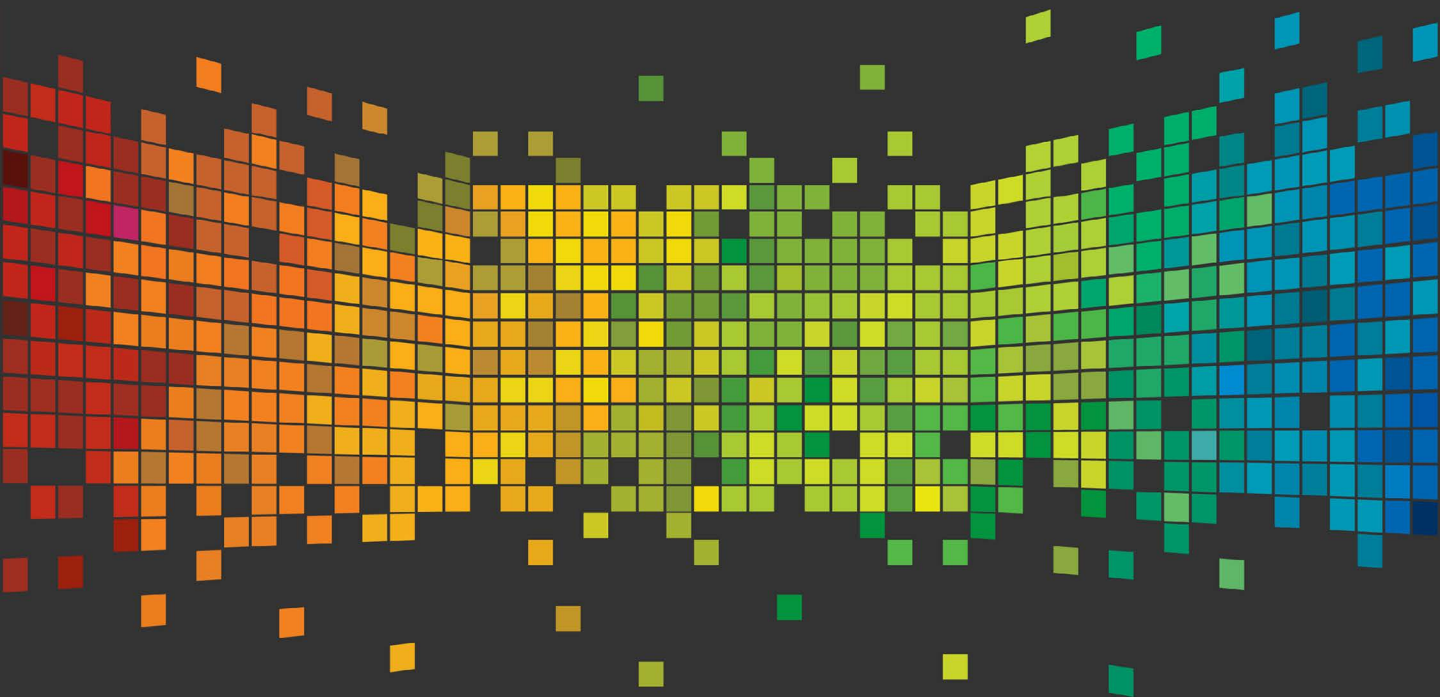
EXPERT INSIGHT

---

# Coding **Roblox** Games Made Easy

The ultimate guide to creating games with Roblox Studio and  
Luau programming

**Second Edition**



---

**Zander Brumbaugh**

**<packt>**

# Coding Roblox Games Made Easy

Second Edition

The ultimate guide to creating games with Roblox Studio and Luau programming

**Zander Brumbaugh**

**<packt>**

BIRMINGHAM—MUMBAI

# Coding Roblox Games Made Easy

Second Edition

Copyright © 2022 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Senior Publishing Product Manager:** Manish Nainani

**Acquisition Editor – Peer Reviews:** Gaurav Gavas

**Project Editor:** Meenakshi Vijay

**Content Development Editor:** Grey Murtagh

**Copy Editor:** Safis Editing

**Technical Editor:** Tejas Mhasvekar

**Proofreader:** Safis Editing

**Indexer:** Manju Arasan

**Presentation Designer:** Pranit Padwal

First published: January 2021

Second edition: June 2022

Production reference: 1300522

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80323-467-0

[www.packt.com](http://www.packt.com)

# Contributors

## About the author

**Zander Brumbaugh** is an independent programmer, project manager, and game designer. The games he has made or contributed to have been played more than 300 million times. He has created popular games including *Anime Lifting Simulator*, *My Salon*, *Munching Masters*, *Power Simulator* amongst others. Zander currently attends the University of Washington and is part of the Paul G. Allen School of Computer Science and Engineering. At the time of writing, Zander is 19 years old.

*To my mother and father. You have both always been there to support me and consistently inspire me to be the best version of myself.*

## About the reviewer

**Harrison Lewis** is an experienced game and software developer who specializes in back-end functionality and security logic. Harrison has over 6 years of experience in the Roblox space, and over 4 years of experience in software development. He has contributed to dozens of Roblox projects and has released a handful of tools and open-source resources.

*I'd like to thank Zander for the opportunity to be the technical reviewer for this book; he's done a wonderful job at making often overlooked resources and techniques easy to understand through his writing.*

# Table of Contents

<b>Preface</b>	<b>xv</b>
----------------	-----------

---

<b>Section I: Introduction to Roblox Development</b>	<b>1</b>
--	----------

---

<b>Chapter 1: Introducing Roblox Development</b>	<b>3</b>
--	----------

---

Technical requirements .....	4
------------------------------	---

Learning the benefits of Roblox development .....	4
---	---

Financial opportunities on Roblox • 4

Improving professional skills • 5

Benefits of networking • 5

Discovering developer types .....	6
-----------------------------------	---

Programmers • 7

Modelers • 7

Builders • 7

UI/UX designers • 7

Gaining a perspective about your early projects .....	8
---	---

Summary .....	9
---------------	---

Worksheet .....	10
-----------------	----

<b>Chapter 2: Knowing Your Work Environment</b>	<b>11</b>
---	-----------

---

Technical requirements .....	12
------------------------------	----

What is an experience? .....	12
------------------------------	----

<b>Traversing the Create page .....</b>	<b>12</b>
Configuring Experience and Place settings • 14	
The Configure Experience menu • 15	
Configure Start Place menu • 16	
<i>Icon • 17</i>	
<i>Access • 17</i>	
Other place options • 19	
<i>Configure Localization • 20</i>	
<i>Create Badge • 20</i>	
The Creator Marketplace and Avatar Shop • 23	
<b>Getting started with Roblox Studio .....</b>	<b>25</b>
The File menu and settings • 26	
Movement and camera manipulation • 28	
Utilizing the Explorer • 29	
Using Studio tools • 32	
<i>The Select tool • 32</i>	
<i>The Move tool • 33</i>	
<i>The Scale tool • 33</i>	
<i>The Rotate tool • 33</i>	
<i>The Transform tool • 34</i>	
Managing the Game Settings menu • 34	
The View tab • 36	
The Test tab • 37	
Customizing Studio to aid your workflow • 40	
<b>Taking advantage of Roblox's resources .....</b>	<b>41</b>
Tutorials and resources • 41	
The Developer Forum and Talent Hub • 42	
<b>Summary .....</b>	<b>43</b>
<b>Worksheet .....</b>	<b>43</b>

---

**Section II: Programming in Roblox** **45**

---

**Chapter 3: Introduction to Luau** **47**

---

**Technical requirements** ..... 48

**Learning about data types and creating variables** ..... 48

    Data types • 48

    Setting and manipulating variables • 51

    Numbers • 52

    Booleans • 53

    Strings • 53

    Tables • 55

    Dictionaries • 57

    Vectors • 59

    CFrames • 61

    Instances • 64

    Conditional statements • 64

**Declaring and using loops** ..... 68

    for loops • 68

    while loops • 71

    repeat loops • 73

**Learning about functions and events** ..... 73

    Functions in programming • 73

    Recursion • 76

    Events and methods of instances • 79

**Demonstrating programming style and efficiency** ..... 81

    General style rules • 81

    Roblox-specific rules • 82

**Summary** ..... 83

**Worksheet** ..... 83



<b>Chapter 4: Roblox Programming Scenarios</b>	<b>85</b>
<b>Technical requirements</b>	<b>85</b>
<b>Understanding the client-server model</b>	<b>86</b>
Different script types	86
Scripts	86
Local scripts	87
Modules	87
The Script Menu tab	89
FilteringEnabled	91
RemoteEvents	92
RemoteFunctions	93
BindableEvents and BindableFunctions	95
<b>Using Roblox services</b>	<b>96</b>
Players service	96
ReplicatedStorage and ServerStorage	98
StarterGui	99
StarterPack and StarterPlayer	99
PolicyService	101
PhysicsService	101
UserInputService	102
<b>Working with physics</b>	<b>104</b>
Constraints	104
Movement constraints	107
<b>Adding peripheral experience aspects</b>	<b>111</b>
Sound	111
Lighting	114
Other effects	117
<b>Summary</b>	<b>118</b>
<b>Worksheet</b>	<b>119</b>

---

**Chapter 5: Creating an Obby** **121**

---

**Technical requirements** ..... 121**Setting up the backend** ..... 122

Managing player data • 123

*Creating a datastore system* • 123*Creating and loading session data* • 125*Manipulating session data* • 127*Saving player data* • 129*Addressing throttling and edge cases* • 130

Managing collisions and player characters • 132

**Making obby stages** ..... 134

Creating part behaviors • 135

Creating rewards • 142

Shops and purchases • 144

Robux premium purchases • 145

Making in-experience currency shops • 152

Preventing exploits • 155

**Setting up the frontend** ..... 155

Creating effects • 156

*Sound* • 156*Particles* • 157*Tying in effects* • 158*Part movement* • 159**Testing and publication** ..... 161**Summary** ..... 163**Worksheet** ..... 163

---

**Chapter 6: Creating a Battle Royale Game** **165**

---

**Technical requirements** ..... 166**Setting up the backend** ..... 166

Managing player data .....	166
Creating weapons .....	168
Setting up the round system .....	181
Preparing the player • 184	
Local replication .....	189
Spawning loot .....	192
Setting up the frontend .....	196
Working with the UI • 196	
<i>Game message and remaining players display • 197</i>	
<i>Making a spectate menu • 201</i>	
<i>Creating a shop • 207</i>	
Summary .....	208
Worksheet .....	209

---

## Section III: The Logistics of Game Production 211

---

### Chapter 7: The Three Ms 213

---

Technical requirements .....	213
Mechanics .....	214
Simulators • 214	
RP games • 216	
Tycoons • 216	
Minigames • 218	
Monetization .....	218
Marketing .....	220
The Roblox promotion system • 220	
YouTubers • 222	
Community • 223	
Reviewing what you've learned .....	224
Chapter 1 • 224	

---

Chapter 2 • 224	
Chapter 3 • 224	
Chapter 4 • 225	
Chapters 5 and 6 • 225	
<b>Metaverse .....</b>	<b>225</b>
<b>Summary .....</b>	<b>226</b>
<b>Worksheet .....</b>	<b>227</b>
 <b>Chapter 8: 50 Cool Things to Do on Roblox</b>	 <b>229</b>
<hr/>	
<b>Technical requirements .....</b>	<b>229</b>
<b>Programming challenges .....</b>	<b>229</b>
Is number x divisible by y? • 230	
FizzBuzz • 230	
Find the maximum value in a table • 230	
Check whether an element exists in a table • 230	
Format seconds into hours:minutes:seconds • 230	
Return unique elements from a table • 231	
Number of stickers • 231	
Concatenate two tables • 231	
Reverse a table • 231	
Sort a table using table.sort() • 231	
Sort a table using a sorting algorithm of your choice • 232	
Solve a linear equation • 232	
Guessing game • 232	
Find the nth number of the Fibonacci sequence • 233	
<b>Experience systems .....</b>	<b>233</b>
Make a leaderboard system • 233	
Make an announcement system • 234	
Make a daily reward system • 235	
Create an interaction system • 235	

- Make a custom ProximityPrompt appearance • 236
- Make a world lighting system • 237
- Make a projectile system • 238
- Make a car system • 238
- Make a racing system • 239
- Make an aircraft system • 239
- Make a ship system • 239
- Make fighting NPCs • 239
- Make a survival system • 240
- Create an inventory system • 240
- Make a pet system • 241
- Make a crafting menu • 242
- Create a house customization system • 242
- Experience ideas ..... 243**
  - Simulators • 243
  - Tycoons • 244
  - Roleplay games • 245
  - Hangout games • 246
- Roblox features ..... 246**
  - Group name changes • 247
  - Free badge creation • 247
  - Spatial voice • 247
  - Mesh deformation • 247
  - Layered clothing • 249
  - Flipbooks • 249
  - Custom materials • 250
  - Talent Hub • 250
- Other development types ..... 251**
  - Plugins • 251
  - UI/UX design • 251
  - Art • 251

<i>Clothing design • 251</i>	
<i>Thumbnails/icons • 252</i>	
<i>Particle design • 252</i>	
<i>Sound and music design • 252</i>	
Animations • 252	
Summary .....	253
<b>Worksheet Answers</b>	<b>255</b>
Chapter 1 .....	255
Chapter 2 .....	256
Chapter 3 .....	258
Chapter 4 .....	260
Chapter 5 .....	261
Chapter 6 .....	263
Chapter 7 .....	264
<b>Other Books You May Enjoy</b>	<b>269</b>
<b>Index</b>	<b>273</b>



# Preface

With this practical book, you'll get hands-on experience working in the Roblox platform.

You'll start with an overview of Roblox development and then understand how to use Roblox Studio. As you progress, you'll learn everything from how to program in Roblox's Lua language, to creating games such as an obby and a battle royale. Towards the end, you'll delve into the logistics of game production, focusing on optimizing the performance of your game by implementing good mechanics, monetization, and marketing practices.

By the end of this Roblox guide, you will have the skills you need to lead or work with a team to bring your gaming worlds to life and extend that experience to players around the world.

## Who this book is for

This book is for anyone who is interested in learning how to develop games on the Roblox platform and those who are already familiar with Roblox and want to explore the best tips, tricks, and practices for efficient Roblox development.

## What this book covers

### Section 1: Introduction to Roblox Development

*Chapter 1, Introducing Roblox Development*, focuses on introducing you to the basic concepts of Roblox development, including getting to know the types of development opportunities that exist, how to make money from games, and what to expect based on earlier projects.

*Chapter 2, Knowing Your Work Environment*, teaches you how to use Roblox Studio. It covers basic controls such as movement and camera manipulation, interacting with instances in the **Workspace**, using free assets, and changing game information.



## Section 2: Programming in Roblox

*Chapter 3, Introduction to Luau*, does not assume any prior programming experience. You will learn how to program in Roblox's Luau language, from `print("Hello world")` to many other universal programming constructs.

*Chapter 4, Roblox Programming Scenarios*, focuses on Roblox-specific programming scenarios that those with general programming knowledge would still be unfamiliar with.

*Chapter 5, Creating an Obby*, is where you will learn how to apply what you've learned to make a simple but neat and playable game. The game type will be an obby game, and to make it, you will need to utilize variables, events, functions, and properties.

*Chapter 6, Creating a Battle Royale Game*, will use everything that you have learned to make a battle royale-style game. This will require you to use all that you have learned in the book while learning new skills in security and organization. By the end of this chapter, you should feel confident in producing your own games from scratch.

## Section 3: The Logistics of Game Production

*Chapter 7, The Three Ms*, is the most comprehensive part of the book that focuses on things outside of programming. This chapter will focus on expanding your skills to make you more than just a programmer and help you market yourself, in addition to optimizing the performance of your games through the three Ms: Mechanics, Monetization, and Marketing.

*Chapter 8, 50 Cool Things to Do on Roblox*, will explore different ideas for games and systems, to inspire you when making games of your own. It will cover programming ideas and challenges, individual systems to add to your games, as well as ways to take advantage of new Roblox features.

## To get the most out of this book

You will require the following hardware and software to implement all the exercises in this book:

Software/Hardware covered in the book	OS Requirements
Roblox Studio	Windows, macOS X, Chrome OS
Roblox Player	Windows, macOS X, Chrome OS

For a complete list of system requirements for all of Roblox's components, please visit the following link: <https://en.help.roblox.com/hc/en-us/articles/203312800>.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

## Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy-2nd-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: [https://static.packt-cdn.com/downloads/9781803234670\\_ColorImages.pdf](https://static.packt-cdn.com/downloads/9781803234670_ColorImages.pdf).

## Conventions used

There are a number of text conventions used throughout this book.

**CodeInText:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example; “When a player joins the game, they need to have their data added to the `sessionData` dictionary.”

A block of code is set as follows:

```
local playersService = game:GetService("Players")
playersService.PlayerAdded:Connect(function(player)
    print(player.Name.. " joined the game.")
end)
```

**Bold:** Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: “To create the user interface for your game, the first thing that you will need to do is navigate to the **StarterGui** service under **Explorer**.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** Email [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at [questions@packtpub.com](mailto:questions@packtpub.com).

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details.

**Piracy:** If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

## Share your thoughts

Once you've read *Coding Roblox Games Made Easy, Second Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.



---

# Section 1

---

## Introduction to Roblox Development

This section will introduce you to the basic concepts of Roblox development. Then we will move on to understand Roblox Studio, the type of development opportunities that exist, and what to expect from your first projects.

This section comprises the following chapters:

- *Chapter 1, Introducing Roblox Development*
- *Chapter 2, Knowing Your Work Environment*



# 1

## Introducing Roblox Development

**Roblox** is a massive entertainment platform like no other, for both playing and creating games. With over 200 million monthly active users, Roblox has enabled both new and experienced developers to create successful games, often called **experiences**, that can be played globally at no cost. With a great number of resources and a large and supportive community of developers from around the world to connect with, entering the Roblox game development scene provides opportunities that cannot be found anywhere else.

Before covering the technical details of working with Roblox Studio and programming in Luau, it is important that you are first familiar with what developing games on Roblox can offer you and what the day-to-day interactions with the platform are like. By the end of this chapter, you will have an understanding of the different roles developers can fulfill on game projects, how you can earn money as a Roblox developer, and what to expect as you develop and publish your first experiences.

In this chapter, we're going to cover the following main topics:

- Learning about the benefits of Roblox development
- Discovering developer types
- Knowing what to expect from your early projects



## Technical requirements

You will not need any software or additional materials for this chapter as it is mostly informative theory. You may want to have access to a computer to visit some of the websites or applications discussed in this chapter.

## Learning the benefits of Roblox development

Roblox has been a consistently growing platform since its creation in 2006, and in recent years, it has grown tremendously. As of 2022, more than 200 million unique users play Roblox monthly. Because of this, there has never been a better time to become a Roblox developer as there are so many new players looking for a wide variety of experiences from new creators.

## Financial opportunities on Roblox

Perhaps one of the greatest contributing factors to Roblox's overall rise in popularity with developers is the monetary benefits that are offered. These benefits may have been one of your motivations to learn more about the platform, and that is to be expected. The top experiences on the Roblox platform are currently making tens of millions of US dollars from in-experience purchases annually.

As a developer, you can earn money from your experiences through the **Developer Exchange**, more commonly called **DevEx**. Roblox has a virtual currency called **Robux**, which can only be purchased with real-world currency. After a player purchases Robux, the balance on their account is updated and then they can spend them freely on any experience they want. When a player spends Robux on one of your in-experience purchases, 70% of the Robux from that purchase will go to you, as Roblox takes a 30% *marketplace fee*.

The two types of in-experience purchases are as follows:

- **Passes**: Single-time purchases
- **Developer products** (more commonly called **dev products**): Purchased multiple times, for something such as in-experience currency

You must have a total of 50,000 earned Robux in your account and be 13 years old or older in order to be eligible for the Developer Exchange program. Earned Robux are those that come from users making in-experience purchases, not Robux you've purchased. In addition to sales you make directly, players of your experiences that have a Roblox Premium membership grant you additional Robux based on how much time they spend playing your experience, though this amount is usually just a fraction of what an experience makes in sales.

Furthermore, there are programs you can apply to that allow you to sell character accessories, plugins, and more for Robux. When working with a development team in a Roblox group, developers can be paid with Robux either directly or through a percentage of experience revenue. Direct pay-outs are a simple action that can be done through the tabs of the **Configure Group** page and are not a recurring payment. Giving developers a percentage of the experience's earnings automatically deposits that amount into their account after it has been verified. Remember, this can only be done when an experience is hosted in a group, not on an individual's profile.



It may be helpful to remember that after a player makes an in-experience purchase, Roblox has up to a 14-day wait period before those Robux are deposited into a group or personal account. This is in order to verify that the purchase was a legitimate sale.

More information about the Developer Exchange program can be found here: <https://www.roblox.com/developer-exchange/help>.

## Improving professional skills

Aside from the monetary benefits of being a Roblox developer, you are developing other skills that you may find applicable to other professional environments. Whether you fill the role of project manager or just an additional programmer, you are able to develop team coordination and communication skills. One of the most sought-after qualities employers look for, especially in STEM, where most work is team-based, is being able to coordinate your work in a team with clear communication. Roblox development is, in my opinion, one of the best places for students in computing-related fields to get their start and learn the basics of collaboration. The platform helps you develop not only greater programming abilities but also strong leadership and eventually money management skills.

## Benefits of networking

While working with other people is not always necessary and there have been popular titles made by single-member teams, the social nature of Roblox strongly encourages developers to work with each other to create experiences, with each developer fulfilling one or more roles in a project. Currently, almost every experience at the top of the **Popular** sort has been created by a team of two or more individuals. With the great success and growth of some top experiences on the platform, the development teams of those experiences have expanded to more than 20 people, though teams of that size are not yet typical.

If you are 13 or older, you can get involved with the community and find other developers to collaborate with through **Twitter**, as well as the communication application **Discord**. By having a Twitter account dedicated to your development work, you can post the creations you are most proud of while communicating with other, more popular developers. These new connections with other developers in the **Roblox Twitter Community (RTC)** may give you the opportunity to collaborate with more well-known individuals and grow your name the more you work. Furthermore, popular Roblox development hashtags like *#Roblox* and *#RobloxDev* let you see what other developers are working on and let your work be seen by others when using them in your tweets. Discord is a communication app that could be compared to **Slack**. There are a variety of Discord servers that are based around Roblox development and are an excellent place to show off creations, discuss your work with other members of the community, and find new people to direct your efforts with. Additionally, you can create your own Discord server to build a following and community around your work. Remember when talking to other people online not to give out personal information and be aware that people may not always be who they say they are.

Another important type of community for developers is **Roblox YouTubers**. As YouTubers create content showcasing your experiences, an impression is made upon their audiences and, as a result, your player count will likely increase from the new publicity. Forming firm connections with these individuals may help you get future promotions for your projects while also creating advocates for your work in the process. While there is not always an easy way to connect with these content creators, they often have a Discord or Twitter presence in addition to business emails where they may respond to you.

Overall, the type of networking described here is the same in many fields of work. Building your image and developing your identity and reputation are the most important parts of your career. Conducting your work in a professional way will have effects that may not be immediately recognized but will certainly be of benefit to you in the future. With better networking abilities, you will become more easily connected with new people who can benefit both your work and reputation.

## Discovering developer types

As previously mentioned, the Roblox developer community is quite diverse; each developer brings their own unique style and technique to the platform. In game development, there are a variety of roles that a developer can fulfill. Most commonly on Roblox, each developer typically has one primary skill that they use for a project. The most common types of developers on Roblox are programmers, builders, 3D modelers, UI/UX designers, and other various artists. Each developer that holds their respective role is equally important to creating a well-refined finished product.

## Programmers

**Programmers** create the core of any experience. From storing player data, creating working weapons, or any other experience functionality that doesn't happen by default, a programmer is the one producing it. On Roblox, programmers use a Roblox implementation of the **Lua** programming language, called **Luau**. Lua is a fast, C-based procedural language that can be found across the game development industry. You will find that the syntax of Luau is less complex than others and seems even more human-oriented than most high-level languages. Because of this, many programmers find the learning curve to be quite low and if Luau is your first language, you may find that the transition to other languages is quite easy as common syntax from multiple programming languages is used.

## Modelers

**3D modelers** create the individual items that you see in an experience, from furniture, pets, food items, to any other visual pieces that are typically small to medium in size. While these models can be made inside of Roblox Studio, most 3D modelers have become skilled in using the free 3D modeling application **Blender** to create their meshes. There are many reasons for doing this; in particular, the **parts** used for creating objects inside of Roblox Studio are quite blocky and you cannot easily achieve smooth or complex shapes, while it is considerably easier to do so in modeling software.

## Builders

**Builders** fulfill the role of creating the worlds that exist in your experiences. Whether it be a cold corridor in a spaceship stuck adrift or a hot, arid desert with an oasis at its center, builders are developers that ultimately create the first impression of an experience when players join. With this in mind, it is important that the builder for a project is skilled enough to create the desired vision. While you may think that builders and modelers are one and the same, they are not. Builders focus more on the overall map and world design for projects, though they often use Blender and simultaneously fulfill the role of 3D modeling for various assets in the world.

## UI/UX designers

**UI/UX designers** create the pages and screens that players interact with inside your experiences. Some good examples of something a UI designer would be responsible for making include a player's inventory screen, a health bar, or any other designed visual interface. Usually, the UI is the first thing a player will notice in addition to the map of your experience; therefore, it would be best that the designer is capable of creating a visually appealing set of UIs that matches the style of the experience itself.

Some additional developer types include animators, music producers, graphic designers, and other artists. All the different types of developers mentioned are important for creating a strong finished product and it is important that they all are equally competent in being able to accomplish the goals of a project. You should be sure to identify which of these roles interest you the most so that you can focus your time on learning more about them and developing those skills for future use.

## Gaining a perspective about your early projects

Though the beginning of your development career may vary, one thing is typically standard: your first experience will not be number one on the popular page, and that's okay! As frustrating as it may be that you are not granted the instant gratification everyone craves, this is the best outcome. Speaking from personal experience, maintaining a popular experience that is played by tens of thousands of players concurrently (or more) is quite stressful. It is best that new developers have time to gain additional experience in order to first get used to the platform before producing popular titles. *Figure 1.1* depicts the thumbnail of my first game, **Endure**; it almost immediately tells an onlooker that the project was made by an amateur:



*Figure 1.1: Endure was one of my first titles and lacked refinement*

Many new developers often run into what I've coined the *Roadblocks on Roblox*, where they begin to create a project that they have a passionate but loosely defined vision for and are forced to abandon it as they struggle to accomplish everything that they had originally imagined. The best way to move past these issues is to lay out a development plan and solidify the features and mechanics that should be included in your project. With this, you can review and restructure your vision as needed, keeping in mind what is popular with Roblox players, as well as what can be realistically accomplished with your personal abilities and that of your team for a project.

Ultimately, motivation and dedication are key; without these, projects simply do not succeed. It is important to remember that games are not often enjoyable to players if they were not enjoyable for the developers to create. If you are collaborating with other people, make sure that everyone on the team is on the same page for the direction and rough roadmap for the development of the game. The best goal to have, for both your mental health and the quality of your projects, is simply to make each new game better than your last. You may spend months developing a game only to have it perform at a mediocre level, but this is part of building your foundation. From each project, you gain a bit more experience and recognition. *Figure 1.2* shows a direct result of following these practices; **Power Simulator** has been played more than 100 million times and its thumbnail is much more engaging to potential players due to its professional level of design:



*Figure 1.2: Power Simulator is one of my most successful titles, a result of years of experience*

No matter what becomes of your early projects, always look forward to the future and try to improve the gaming experience for your players while improving the development process for yourself; only with repeated effort will you find success.

## Summary

The key points to remember from this chapter are what opportunities exist on the Roblox platform, what different developer types do, and what interests you, as well as finding a collection of people in the community with which to create more well-rounded projects.

After combining what you have learned about networking, what to expect from early development, as well as how to build your overall experience as a developer, I believe that you too will be able to accomplish amazing feats on the platform while strengthening skills that will benefit you in any professional environment.

In the following chapter, you will begin getting to know Roblox Studio, the program with which you will create your experiences. Knowing all the features of your work environment, including those that may not be immediately noticeable, will help to increase your development efficiency and overall productivity on future projects.

## Invitation to join us on Discord!

Read this book alongside other users and the author.

Ask questions, provide solutions to other readers, chat with the author via *Ask Me Anything* sessions and much more. Scan the QR code or visit the link to join the community.



<https://packt.link/letscreategames>

## Worksheet

- Q1. What is a common and interchangeable word for games used on Roblox?
- Q2. What is the name of the process through which developers exchange Robux for real-world currency?
- Q3. What do developers sell for Robux in their games in order to earn money? Specifically, what are used for one-time purchases and what are used for multiple purchases?
- Q4. How long may it take for developers to receive the proceeds of in-experience sales?
- Q5. What is the name of the programming language used in Roblox development?
- Q6. What do programmers do on Roblox?
- Q7. What do 3D modelers do on Roblox?
- Q8. What do builders do on Roblox?
- Q9. What do UI/UX designers do on Roblox?
- Q10. What should you expect from your first projects on Roblox?

# 2

## Knowing Your Work Environment

Fully knowing your work environment allows you, as a developer, to make progress on your projects more efficiently. Familiarizing yourself with the interfaces that Roblox developers work with on a daily basis in order to produce expansive, high-quality experiences will set you on the path to making your own.

After reading this chapter, you will be able to create new **experiences**, change external experience settings, navigate and use **Roblox Studio**, and make use of the additional resources Roblox offers. You will also learn how to fine-tune some of the smaller yet convenient features it provides.

In this chapter, we're going to cover the following main topics:

- Traversing the **Create** page
- Getting started with Roblox Studio
- Exploring plugins and the **Toolbox**



## Technical requirements

For this chapter, you will need access to a computer that meets the requirements to run Roblox Studio. In order to use Roblox Studio, you will also need an internet connection. More information on the system requirements of Roblox can be found here: <https://en.help.roblox.com/hc/en-us/articles/203312800>.

## What is an experience?

Roblox has long been a gaming platform but with the concept of the **metaverse** on the rise, Roblox changed its focus to become broader. As a result, a *game* became an *experience*, a *player* became a *user*, and many other changes like this occurred. An experience is anything that you can imagine creating, be it social, casual, realistic, or abstract in nature. In the next section, we will look at the parts of the Roblox website related to creating experiences.

## Traversing the Create page

The **Create** tab of the Roblox website displays all of your created experiences and uploaded assets, showing the most recently created first, in the tab related to the asset's category. Additionally, many of the actions that can be taken to change the different overall settings of your experience can be completed here. To access the **Create** page, navigate to <https://www.roblox.com/develop>, and you will be brought to the page depicted in *Figure 2.1*:

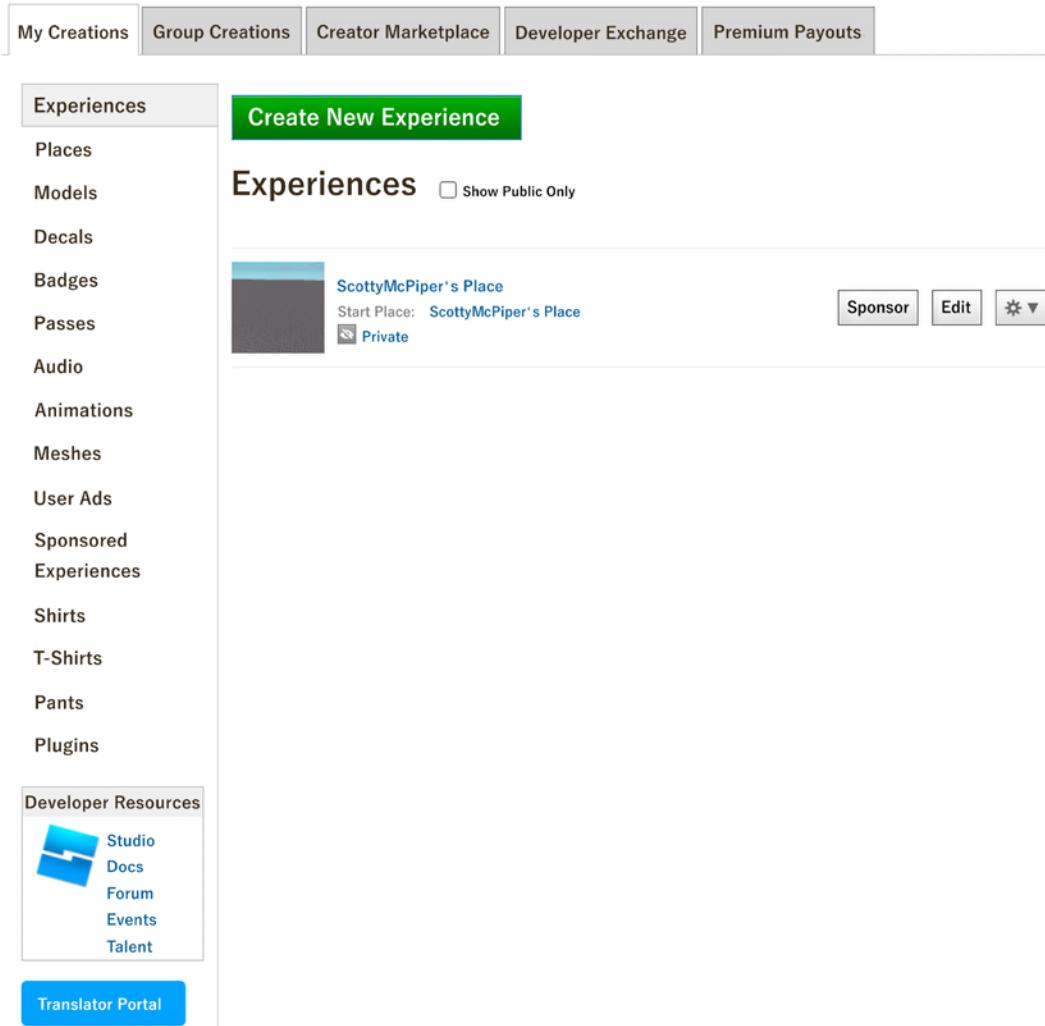


Figure 2.1: The Create page shows everything related to development



We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: [https://static.packt-cdn.com/downloads/9781803234670\\_ColorImages.pdf](https://static.packt-cdn.com/downloads/9781803234670_ColorImages.pdf).

To continue with this chapter, you need to navigate to the **Create** tab. Once there, find the default experience that was created when your account was activated.

If for some reason no experiences are shown in a list under the **Create** tab, click the **Create New Experience** button. This should open Roblox Studio. From there select a new experience template; I suggest choosing **Baseplate** to start. Once in the template, press **Alt + P** and then the **Create** button. By pressing **Alt + P** you *published* your game to Roblox; this will be covered in more detail in the *Getting started with Roblox Studio* section. We'll learn all about how to use Roblox Studio soon, but at this point, you can return to the **Create** page and continue.

Now you are all set to learn how to change the external settings of your experience and make use of the other resources that the **Create** page offers.

## Configuring Experience and Place settings

Perhaps you have realized that there are certain settings you will want to change for your experience, such as how many visitors can fit into a server or even what platforms the experience can be played on. For settings such as these, no programming is needed; the changes can, and in most cases should, be done through the different options accessible via the **Create** page.

Once you are under the **Experiences** section of the **Create** page, which is selectable on the left side of your screen, click on the *gear* icon on the right side of your experience, as seen in *Figure 2.2*. By clicking this icon, you will be presented with a drop-down menu that lists a variety of choices:

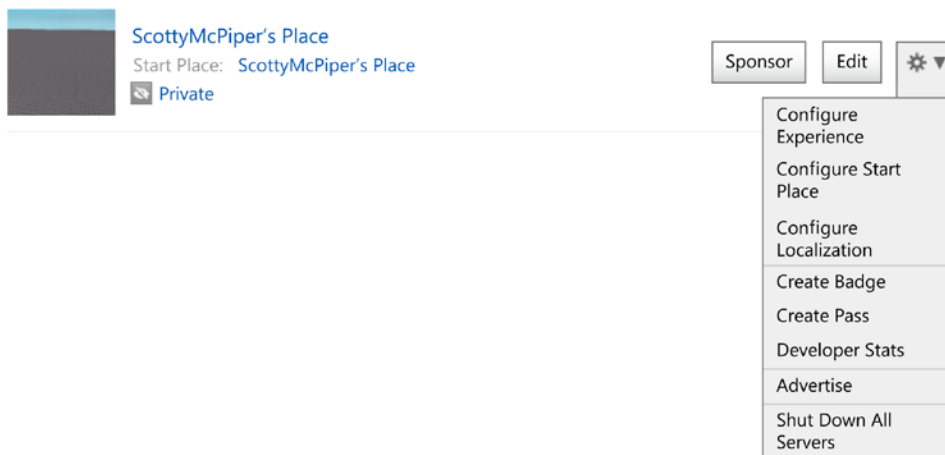


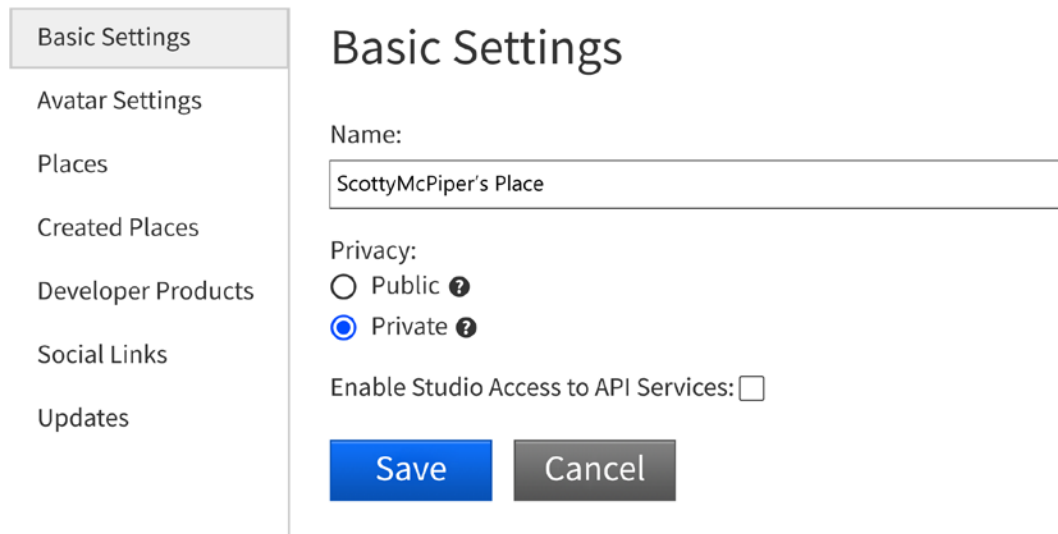
Figure 2.2: A default view of an experience's gear icon menu on the Roblox website

For this section, however, the focus will be on the **Configure Experience** and **Configure Start Place** options, as there is much important information to cover.

## The Configure Experience menu

By clicking **Configure Experience**, you will be able to make a number of changes to your experience. You will notice the ability to change your experience between **Public** and **Private** privacy statuses, as shown in *Figure 2.3*:

### Configure Experience



The screenshot displays the 'Configure Experience' interface. On the left, a vertical sidebar lists several settings categories: 'Basic Settings' (highlighted), 'Avatar Settings', 'Places', 'Created Places', 'Developer Products', 'Social Links', and 'Updates'. The main content area, titled 'Basic Settings', includes a 'Name' input field containing 'ScottyMcPiper's Place'. Below this is a 'Privacy' section with two radio button options: 'Public' and 'Private', with 'Private' being the selected option. A checkbox for 'Enable Studio Access to API Services' is present and unchecked. At the bottom of the settings area are two buttons: a blue 'Save' button and a grey 'Cancel' button.

*Figure 2.3: The Configure Experience menu is one page where external experience settings are changed*

Let's look at the difference between the **Private** and **Public** settings:

- When an experience is set to **Private**, no users except yourself and those you manually give permission to will be able to join your experience.
- When an experience is set to **Public**, all Roblox users will be able to join your experience. We will discuss how to change these permissions for specific users once we get to the topic of Roblox Studio.

Both the **Configure Experience** and **Configure Start Place** pages have a tab where dev products can be created and configured. While the process is straightforward enough to be self-explanatory, note that you can add a name, price, description, and image for your dev products. When we cover creating your own experience, we will discuss how to link the purchase of both passes and dev products to your scripts, in order to make them function properly.

Further down the list of tabs for the **Configure Experience** page, Roblox allows you to place links to different social media accounts. This is the only place where you should include social media links, as Roblox states in its terms of service that it shows or hides certain social media links based on the age listed on a user's account in order to protect younger users. As has been found by many unfortunate developers before, a violation of this clause may lead to moderation action against your account or experience.

## Configure Start Place menu

Now, if you click on **Configure Start Place**, you will find many more options that are relevant to the appearance and settings of your experience as a whole; some of these settings are visible in *Figure 2.4*. You are immediately presented with the **Basic Settings** tab, which allows you to change the name of your experience and the description that is shown beneath it:

### Configure Place

Basic Settings

Icon

Thumbnails

Access

Permissions

Version History

Developer Products

Experiences

### Basic Settings

Name:

ScottyMcPiper's Place

Description:

If you have built [Premium benefits](#) into your experience, please list those benefits in the description.

⚠ Updating the start place's name or description will also apply to the experience.

Genre:

All

Save Cancel

*Figure 2.4: The Configure Place page is like the Configure Experience page but has different settings*

## Icon

The second tab, titled **Icon**, allows you to change the **icon** of your experience. An icon is a square image that typically serves as the first impression users have of your experience when seeing it on the **Discover** page or on a profile. Similarly, the **Thumbnails** tab allows you to upload images that represent your experience and serve as the second and final impression before a user clicks the join button for your experience. For 500 Robux, you can also upload a video up to 30 seconds in length; this feature allows you to show your experience in action, which is often more engaging to potential visitors.

## Access

The **Access** tab allows you to change different options for who can join your experience. You will first see a list of checkboxes for the different platforms your experience can be joined from as seen in *Figure 2.5*. When we get to creating your first experience, you will see that you should enable or disable these options based on the mechanics of your experience and overall cross-platform compatibility for controls. Below these boxes, you will see an option to sell access to your experience. While most experiences are free to join on Roblox, you can charge Robux for users to access your experience (but in most cases, this will lower the number of people who join your experience).

## Configure Place

Basic Settings

Icon

Thumbnails

Access

Permissions

Version History

Developer Products

Experiences

### Access

Playable devices:

☒ Computer

☒ Phone

☒ Tablet

☐ Console

Experience Access:

☐ Sell access to this experience

Maximum Visitor Count:

?

Server Fill:

☒ Roblox optimizes server fill for me ?

☐ Fill each server as full as possible

☐ Customize how many server slots to reserve ?

Access:

▼

Private Servers:

☐ Allow Private Servers ?

Figure 2.5: The Access tab has settings for what type of, and how many, users can join your game

The next setting, **Maximum Visitor Count**, should be changed based on how many visitors should realistically be able to join your experience on one server. While the decision for this value is up to you, remember to consider the performance limitations of your experience as well as how its various mechanics may be affected if there are too many or too few users available. For example, if you are making a social experience, there should be enough users to make it feel lively rather than empty.

A setting that is very closely related to **Maximum Visitor Count** is how Roblox decides to fill servers with users. By default, the **Roblox optimizes server fill for me** option is selected, and it is typically best to leave it as such. It may, in some situations, be a better choice to reserve some server slots if you find your experience benefits greatly from users having their friends join them, but as a beginner, there is no reason to change this setting.

The **Access** setting has the option of **Everyone** or **Friends/Group Members**, depending on whether the experience is being hosted on a user's profile or in a group. You may change this setting at your own discretion, but like before, if your experience has restrictions regarding who can join, then your visitor count will be limited by that factor.

The last setting of the **Access** tab allows you to enable the sale of **private servers** (formerly called **VIP servers**). Private servers are a monthly subscription purchased by users so that they can visit your experience uninterrupted, only with friends or those that they invite. Since private servers are a recurring purchase, they can continue to generate passive income even after an experience declines in popularity.



In order to sell access to your experience, you cannot also be selling private servers. When selling access to your experience, the minimum price for access is 25 Robux and the maximum is 1,000 Robux. When selling private servers, the minimum price is 10 Robux if you do not want them to be free. There is currently no limit to the maximum price of private servers.

While there are some other tabs of the two different settings pages that were not covered in this section, most of them have been deprecated in favor of ones accessible from Roblox Studio or are settings that you would typically not need to interact with under normal circumstances. Once you become a more experienced developer and you find that you need to change those options, you will know how to. Next, we'll explore other experience settings you can change.

## Other place options

Returning to the **Create** page, there were many more options on the gear drop-down menu previously shown in *Figure 2.2*; you should be familiar with these before creating your first experience. While some of these topics will not be covered in depth, you should know what they are in order to have an advantage when you begin developing more intricate projects.



## Configure Localization

Immediately following the **Configure Experience** and **Configure Start Place** options, you will see a **Configure Localization** option. This page allows for your experience to be manually translated into different languages. From here, you can add different Roblox users or groups to translate your experience for, add or remove languages you want your experience to be supported in, and see what text entries from your experience have already been recorded and translated. Roblox is also in the process of releasing automatic translation for both experience web pages and in-experience content. For more information on configuring localization, see the primary article on Roblox localization here: <https://developer.roblox.com/en-us/articles/Introduction-to-Localization-on-Roblox>.

## Create Badge

The next option on the menu is **Create Badge**. **Badges** are items that are awarded to users and show up on their profile. Once awarded to a user, a badge cannot be given again unless the user deletes it from their inventory. While badges are mostly used for display or novelty purposes, they are sometimes used as somewhat of a hacky way to track different data types. For example, if you want all users who tested your experience pre-release to receive something special later on, the most convenient way to do that might be to simply award them a badge. With this technique, even though user data will likely be reset when the experience releases, you can still identify whether a user is in possession of that badge and subsequently award them for being an early supporter of your project.

The **Create a Pass** selection will bring you to a different tab of the **Create** page. Like dev products, you can add a name, description, and image to your pass with the interface depicted in *Figure 2.6*:

## Create a Pass

Target Experience: [ScottyMcPiper's Place](#)

Find your image:  No file chosen

Pass Name:

Description:

*Figure 2.6: The interface for creating passes shown here is similar to that of dev products*

Once created, your pass will be listed, and clicking the *gear* icon next to it will allow you to configure the price of the pass, whether it is on sale, and edit its name, description, and image.

The **Developer Stats** tab will become quite important once you begin monitoring the performance of your own experiences. The page this button brings you to is the center for all of your experience analytics. From here, you can see the average engagement time of a given user, the total amount of Robux earned from in-experience sales before tax, and how many users are playing on which platforms, all viewable as datasets over different time intervals. In addition, you can export the analytics of your experience as a CSV or XLSX file for a further breakdown of your data in your desired external application.

In order for your experience to reach users, Roblox has two built-in types of promotion, called **User Ads** and **Sponsored Experiences**. Promoting your experience through these systems costs Robux and the more you bid, the more impressions on users your promotion will have (see *Figure 2.7*). You have the choice for which platforms your campaign runs on, meaning if your experience is not available to join on Xbox, you needn't waste money by showing promotional information to Xbox users. Furthermore, you can choose which age and gender demographics you wish your experience to be shown to.

So, if you think your experience will be more popular with users of a certain age group or gender, you can focus your spending on those groups:

## Sponsored Experience Ad Creation


### Experience

Choose which ScottyMcPiper experience you will advertise.

ScottyMcPiper's Place ▼

### Icon

Your ad will always show the current version of your icon.



### Targeting

Choose who can see your ad.

Gender ☒ Any ☐ Female ☐ Male

Age ☐ Under 13 ☐ 13+

Platform ☐ Phone ☐ Tablet ☐ Computer ☐ Console

At least one from each targeting type must be selected.

### Schedule

Choose the duration of your ad.

1 ▼ Days

Currently you can only start an ad today, and it can run for a max of 28 days.

### Daily Budget

How many Robux do you want to spend per day?

Robux per day

The higher your daily budget, the more impressions your ad is likely to get.

### Summary

Total Budget: 0 ?

Schedule: 02/13/2022, 01:43 PM - 02/14/2022, 01:43 PM

### Ad Name

Choose a descriptive name for your ad.

0/50

This ad name is only visible to you.

[Preview Your Ad](#)

Figure 2.7: The more Robux you bid, the more estimated impressions you will receive

The difference between advertising and sponsoring an experience is simply how your promotional material will physically appear to potential visitors. For example, when using the **Sponsored Experiences** system, the icon of your experience will show wherever a list of experiences is present, including on the front page and when a user searches for experiences with the search bar. Underneath your icon, the words **Sponsored Ad** will be displayed. When using the **User Ads** system, you must upload an image of a certain size from a list of specified dimensions. While sometimes User Ads yield a higher **Click-Through Rate (CTR)** (clicks divided by impressions) than Sponsored Experiences, they are less often displayed to users. The reason for this is because most ad-blocking plugins for web browsers automatically hide them and, by default, ads are no longer shown on the **Discover** page. I personally use the **Sponsored Experiences** system, though it is a current debate within the developer community which system is better. The effectiveness of your choice will ultimately depend on how engaging the design of your icon or advertisement image is.



You may find it valuable to know that the number of impressions you receive scales linearly with how much you bid, meaning that if you were to bid twice as much, you would receive approximately two times as many impressions. How many impressions you receive from a promotion will vary daily based on how many other developers are promoting their experiences. For additional information on promoting your experiences, visit this page: <https://developer.roblox.com/en-us/articles/Advertising>.

## The Creator Marketplace and Avatar Shop

The **Creator Marketplace**, formerly called the **Library**, is a collection of experience assets created by other users; assets are anything uploaded to Roblox, be it clothing, a model, an image, etc. Better known as the **Toolbox** when using it directly from Studio, developers can find a wide array of free scripts, audio, models, images, meshes, and plugins that can be used in their experiences for free, without needing to give credit. While it is an excellent resource that should be taken advantage of, there are some things to keep in mind when using it.

Firstly, make sure that you are not overusing free models. While there is no shame in doing so, other members of the development community often look down on the overuse of them, and more importantly, your projects consist of less of your own work. Additionally, while most models are completely safe to use, some malicious users take pride in placing scripts in these models that can cause lag or unexpected behavior in your experience. This vulnerability does not exist when searching under the image, audio, or mesh categories, as scripts cannot be added to these assets in the **Toolbox**.

The best advice is to use free models in moderation: look to see whether they are positively rated, and search through them for undesired scripts before use.



As a note of caution, if you decide to upload your own assets to the **Toolbox**, make sure that they are appropriate and abide by the moderation guidelines. Uploading inappropriate or, in some cases, copyrighted assets can lead to a moderation action against your account. This could range from the deletion of the asset to a ban from the website, depending on the severity of your offense.

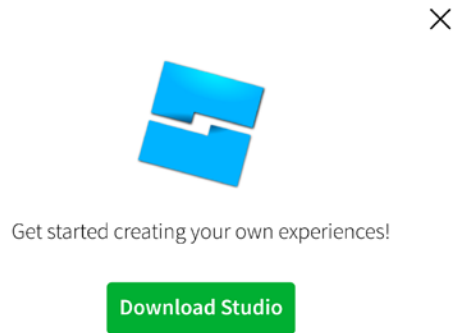
The **Avatar Shop**, formerly and more commonly called the **Catalog**, is a collection of clothing and accessories for Roblox characters. Most accessories are made by Roblox, but some are made by reputable members of the community that have been accepted into the **UGC Program**, **UGC** standing for **User-Generated Content**. For example, clothing can be made by anyone but may cost Robux to upload depending on the clothing type. These clothing items and accessories can be used for free in your experiences either as part of character design or, in some cases, map additions.

You can now completely utilize the **Create** page in order to make experiences of your own. Some important skills you have learned include how to create new experiences and fully configure their external settings, promote your experiences so that they can be found by more potential visitors, and take advantage of free assets to ease your development process. These will all contribute to your ability to efficiently make a successful project as you learn how to use Roblox Studio in the following sections and begin programming in the following chapters.

## Getting started with Roblox Studio

**Roblox Studio** is the application in which you will create all of your experiences. With a variety of tools, information, and add-ons, Studio will function as your development hub from the conception of a project to its release.

Return to the **Create** page and click the **Edit** button next to the *gear* icon. If Studio is not already downloaded, you will be prompted with a set of instructions to download it, as seen in *Figure 2.8*; the program will launch automatically after installing:

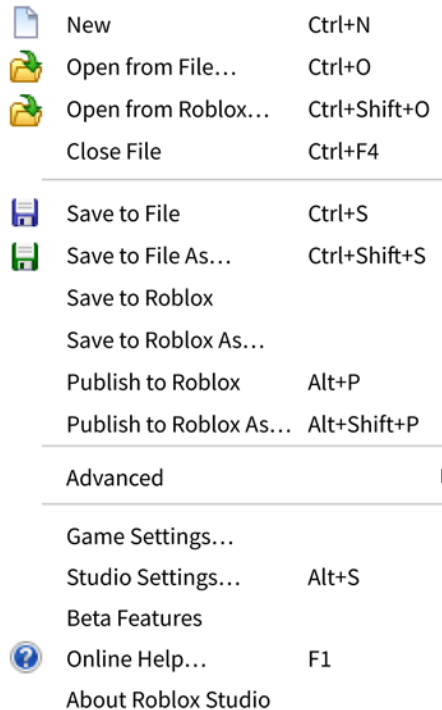


*Figure 2.8: Clicking the Edit button will prompt you to download Studio if it is not installed*

By the end of this section, you will know how to navigate through the tabs and menus of Roblox Studio, be knowledgeable about the various tools within Studio, and be ready to begin setting up your first experience.

## The File menu and settings

As in most applications, the **File** menu holds a variety of actions and submenus that are a core component of your use of the application; some of these are shown in *Figure 2.9*. While not all of these may be relevant to a beginner developer, there are some that are necessary for everyday Studio interactions:



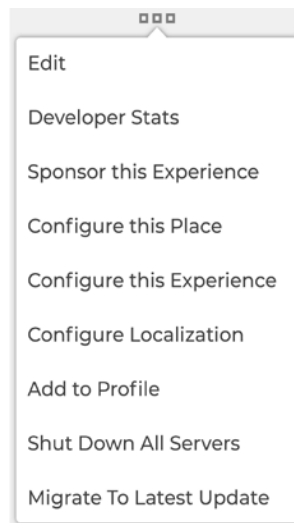
*Figure 2.9: A view of the Roblox Studio application's File menu.*

Starting from the top of the list, the **New** option simply opens a new Studio window with only a baseplate inside of the **Workspace**; a baseplate is essentially a large block that you can work upon. Comparably, the **Open from File...** and **Open from Roblox...** options will open a new Studio session and prompt you to select an existing place accordingly. The **Close File** option will not close the window but instead bring you to a similar page as the **Open from Roblox...** action, which shows you a list of your created experiences.

The **Save to File** and **Save to File As...** selections are used when you want to create or update a place file. Place files use the `.rbxl` extension; these files are usually quite small and can be easily transferred.

The **Save to Roblox** and **Save to Roblox As...** actions function similarly but will save your experience to Roblox's servers.

**Publish to Roblox** and **Publish to Roblox As...** are a critical part of updating your experience. As you make edits to your experience, you will often notice autosaves, but they will not be pushed to affect new servers; instead, you must **publish** your experience to Roblox whenever you make new changes that you wish to be in your experience. Something to remember is that when you successfully publish the current version of your experience, it will not affect live servers that are already running. Instead, you must shut down all of the servers of your experience or gradually migrate them to the current version. You can shut down the servers of your experience by navigating to its web page and clicking **Shut Down All Servers** from the menu depicted in *Figure 2.10*. This menu can be found on the web page of your experience by clicking the three dots at the top right:



*Figure 2.10: The three dots to toggle this menu are visible on all experiences you can edit*

It is important that you know that visitors will be kicked out of your experience and forced to rejoin when you use this option. The **Migrate To Latest Update** alternative also kicks out visitors but minimizes user rejoin times and overall disruption by creating new servers while shutting down old ones over a 6-minute interval rather than all at once. This is the optimal action for updating if you are not publishing a hot-fix, that is, something that must be fixed urgently, or anything else that does not need to be immediately available to users.



The **Online Help...** button will bring you to the Roblox website, intended to aid developers. This website shows not only tutorials but also an abundance of documentation and other assistive assets that we will cover later on in this chapter.

By clicking the **Settings** buttons, a new dialog box will appear within Studio. There are many settings, some of which are quite specific and advanced. Most of the options here do not need to be changed by a beginner except for some settings that you may simply want to change for the sake of convenience; those settings, which are mostly cosmetic, will be explained in the following sections.

## Movement and camera manipulation

Manipulation of your camera in Roblox Studio is quite simple. Your camera will move relative to the direction you are facing:

- The *W* key will move your camera forward
- The *A* key will move your camera left
- The *S* key will move your camera back
- The *D* key will move your camera right
- The *E* key will move your camera up
- The *Q* key will move your camera down
- Additionally, holding the *Shift* key while moving your camera will cause movement to occur at a slower rate, which is convenient for examining smaller objects in the **Workspace**

In order to change the direction your camera is facing, hold down the right button of your mouse and move it. Using your scroll wheel, you can move your camera by a set amount. By default, the scroll wheel will move your camera in the direction of your mouse position on screen, but this can be changed to simply increment toward or away from the direction your camera is facing by unchecking the **Camera Zoom to Mouse Position** setting in Studio.

If you are not satisfied with your current camera speed, it can be altered by navigating to the **Studio** tab of the **Settings** dialog box and changing the related values under the **Camera** section, as shown in *Figure 2.11*:

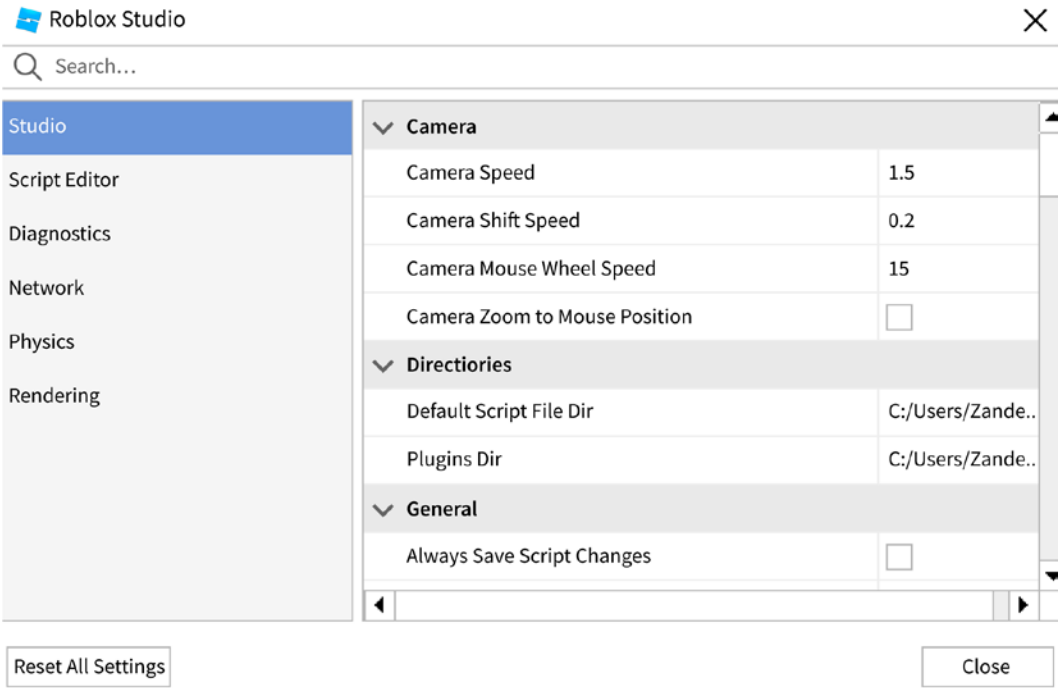


Figure 2.11: Many settings for Studio can be found under the Studio tab of the settings menu

While it depends on the performance of your system, you may also want to increase the graphics settings of Studio. To do this, navigate to the **Settings** dialog box and access the **Rendering** tab. At the top of the **Rendering** tab, the **Quality Level** and **Edit Quality Level** settings will be set to **Automatic** by default. It is recommended that you change these values to **Level 21**, where all the details of the **Workspace** will be visible.

## Utilizing the Explorer

By navigating to the **Home** tab of Studio and clicking the **Part** button, you will see the part, which is basically a block, appear in front of you. By selecting the part and pressing the *F* key, your camera will focus on the part; this works for any physically selectable object. On the top-right side of the default Studio screen, you will see the **Explorer**. The **Explorer** serves as an interface that allows you to see all of the objects, more commonly called **instances**, within your experience. We will cover **instances** and other objects that show up in the **Explorer** in much more depth in the following chapters.

For now, you will see the **Workspace** option at the top of the **Explorer**; it is depicted by a globe, as highlighted in *Figure 2.12*:

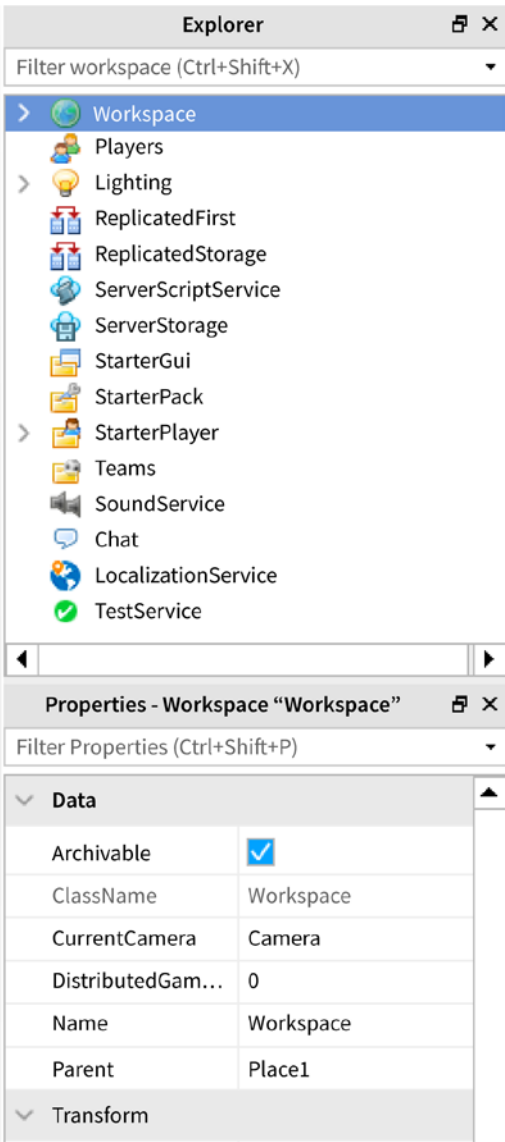


Figure 2.12: This image shows the Explorer and Properties menus with Workspace selected

After selecting the part, you can see that there are now many settings for it shown under the **Properties** menu. Properties of instances control how they behave or appear to your visitors.

For parts, **Anchored**, **CanCollide**, **Transparency**, **Material**, and **Color** are some of the most commonly changed properties, and they can be seen in *Figure 2.13*:

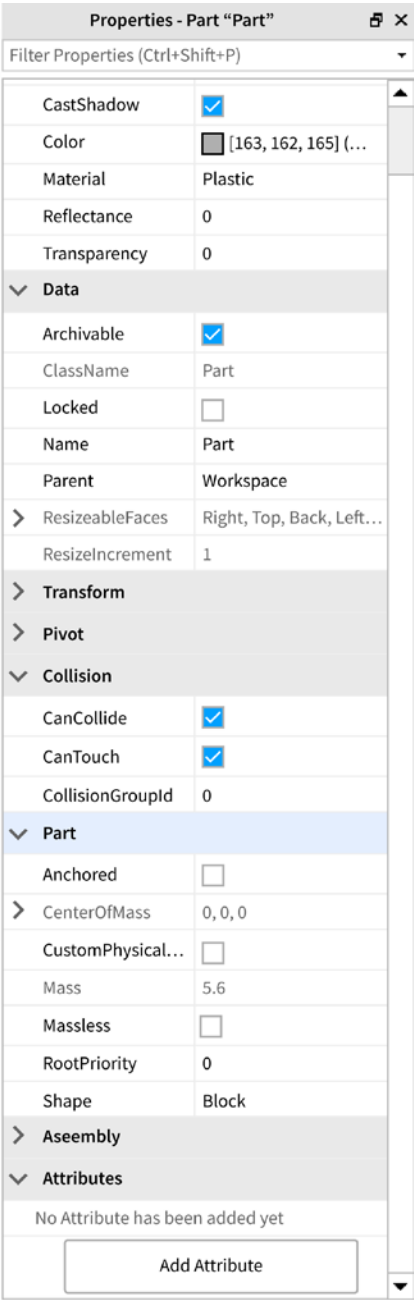


Figure 2.13: Here, the Properties menu shows the discussed properties of the Part instance

Let's look at those five features:

- When a part is set to **Anchored**, it cannot be moved by gravity or any other physics-related forces; a part with a false anchored property is called unanchored.
- The **CanCollide** property determines whether a part can collide with any other instance in the **Workspace**. When disabled, the part will fall through the map until it is destroyed. Because of this, you will want your part anchored if this setting is enabled.
- The **Transparency** setting allows you to change how visible a part is and has a range of 0 to 1.
- **Material** is a list of different textures for parts that also have different visual and physical properties, including reflectivity, density, elasticity, and friction, though these can be changed via the **CustomPhysicalProperties** property.
- The **Color** setting simply allows parts to have their color changed; RGB and HSV values are supported. Roblox also includes a menu of titled, preset colors under the **BrickColor** property.

**Model** is another type of instance, one that has the primary use of grouping different instances together. By holding the *Ctrl* key, you can select multiple parts either through the **Explorer** or by clicking on them in the **Workspace**. After selecting multiple parts, you can press *Ctrl* + *G* to do a group action, parenting all of the selected instances into a new model. This is convenient for organization as well as for use with some tools, which will be explained in the following section.

## Using Studio tools

Roblox provides several *tools* within Studio so you can manipulate objects in the **Workspace**. We'll cover them all in this section.

### The Select tool

The **Select** tool works simply as a dragger for items in the **Workspace**; clicking and holding on an instance in the **Workspace** and moving your mouse will cause that object to go to wherever your mouse hits. One option you should disable when working with anchored parts is **Join Surfaces**, as it creates additional, unneeded objects in whatever is being dragged. This feature can be useful when working with unanchored parts, though, as it will connect them together but they will still be affected by physical forces. The button for this can be found to the right of the four tools under either the **Model** or **Home** tab.

## The Move tool

The **Move** tool shows six handles through which to move the selected object. These handles point in directions relative to the object by default, to the front, back, top, bottom, left, and right. Clicking and holding on these handles and dragging along the chosen axis will allow precise movements along said axis without changing position on the others. Some of the less obvious features you may want to be aware of when using this tool are the **world coordinate** toggle, the **Snap to Grid** setting, and the **Collisions** setting. By pressing *Ctrl + L*, the handles will point in global directions, instead of those relative to the instance. That is to say, the handles will always point up, down, right, left, forward, and back regardless of the object's orientation. This feature is particularly important when an object is rotated, and you want the **Move** tool to behave as if it were not rotated. The **Collisions** setting is on by default and prevents you from moving objects through one another. While this may be convenient in some circumstances, you will most likely want to have this option disabled, as it prevents you from moving parts when even the slightest collision with another part occurs. The **Snap to Grid** setting is under the **Model** tab and allows incremented control over the movement of your part. You can keep this on if you would like, but many developers tend to have this set to 0.1 studs (the Roblox unit of measurement) or some other small increment in order to have free movement over parts and models in the **Workspace**.

## The Scale tool

The **Scale** tool allows you to scale both models and parts in size. Much like the **Move** tool, you can utilize the **Snap to Grid** feature when scaling instances. The handles for the **Scale** tool will always be relative, so the world coordinate toggle will have no effect. Two additions to the scale tool are the **mirrored scale** and **whole scale** behaviors. By holding the *Ctrl* key while scaling a part, the scale action will be mirrored along the other side of the selected axis. Moreover, while models always scale as a whole, meaning they maintain their relative dimensions, you can scale a single instance with the same behavior by holding the *Shift* key while scaling.

## The Rotate tool

The **Rotate** tool also has three axes, X, Y, and Z, that you can rotate an object around. The **Rotate** tool takes advantage of both the world coordinate toggle and the **Snap to Grid** option. The **Snap to Grid** option input box for the **Rotate** tool is located above the one used for the **Scale** and **Move** tools and takes the desired change in degrees as opposed to change in studs.

## The Transform tool

The **Transform** tool serves as a combination of the aforementioned four tools and is located under the **Model** tab. It may be used at your preference and has some advantages, including the ability to freely utilize the functionality of all four tools without being required to switch between them. Conveniently, the tool provides you with a grid along the bottom of the selected instance to guide you.

## Managing the Game Settings menu

So far, you have learned how to change many settings for your experience through the **Create** page, but Roblox includes some experience options that can only be changed in Studio. The function of the **Game Settings** menu is similar to the **Create** page but exists primarily to show additional settings and to provide convenient access for previously mentioned ones. *Figure 2.14* shows the appearance of the **Game Settings** menu and its tabs:

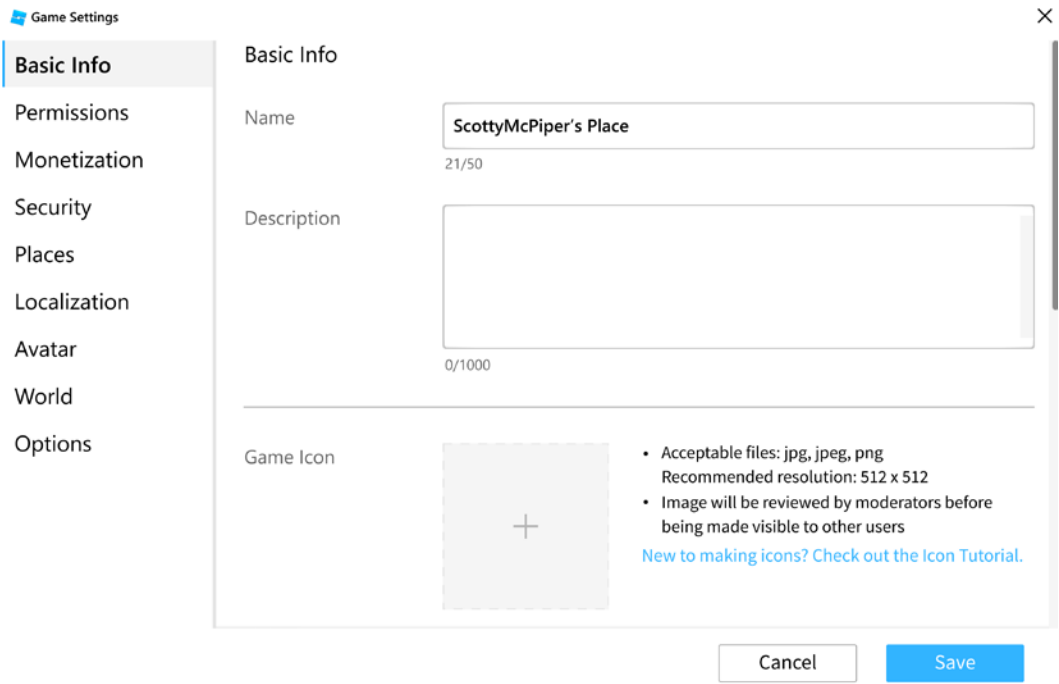


Figure 2.14: The Game Settings menu allows you to change settings on the Create page and more

The **Permissions** tab of the menu is useful for managing who can access your experience. Previously, if you wanted to have only a select few people test your experience, they would have to be ranked as a developer, or your testers would have to be the only members in a group and your experience restricted to only be playable by group members. With this menu, you can set permissions for different users or group roles to join or edit your experience. By granting the **Edit** permission to a user or role, you can take advantage of the **Team Create** feature, which will be covered in the *View tab* section.

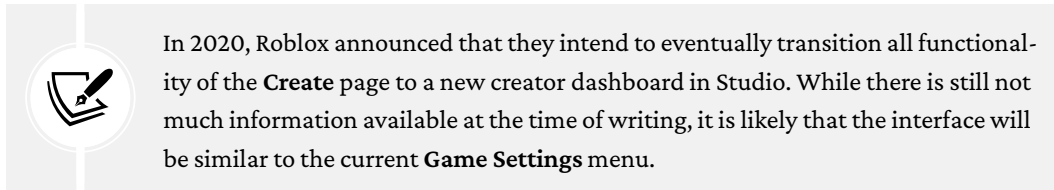
The options and presets visible under the **Avatar** tab were previously under the **Configure Experience** menu of the **Create** page but were relocated for ease of access and due to the fact that they have more direct relevance to Studio. As you will see here and in some of the following tabs, you can leave most of the settings alone as the Roblox default values will likely suit most of your creations. However, if you are not familiar, Roblox has two main character types: **R15** and **R6**, with the number in each name signifying how many parts make up the character model. You will find, when programming, that you may want to index a body part that only exists in one of these rigs, such as a hand, for example. In order to limit what you need to check, you can force which character type is used in your experience in addition to some scaling and animation choices. While most of the behaviors accomplished by these settings can be done by your own programs, it is best, in most situations, to use this menu for the sake of ease.

The **Options** menu provides you with the ability to enable **Collaborative Editing**. This feature serves as a form of source control and is quite convenient when multiple programmers are working at once. Source control is a system for managing different versions of and changes to code sources; this becomes very important when there are multiple programmers at work on the same project. When editing a script with this option enabled in **Team Create**, a **Drafts** button will appear under the **View** tab. Alterations made will only occur locally until the script has been committed; if edits to the program have already been made, you can observe the differences and merge before committing. While this feature does not contain some of the desired aspects of source control that something like **Git** offers, it can be used similarly with no additional setup.

The **Security** tab is a crucial part of developing a new experience with programmed systems. For example, if you wish for your experience to be able to communicate with remote servers or allow Roblox datastores to save data while you are in Studio, those options will need to be enabled in this menu. The two other settings you will see are third-party sales and third-party teleports. If you program your experiences with even the most minimal security, which you will be taught to do throughout *Chapter 5, Creating an Obby* and *Chapter 6, Creating a Battle Royale Game*, you can have these settings enabled as they may be of use in some future scenarios.

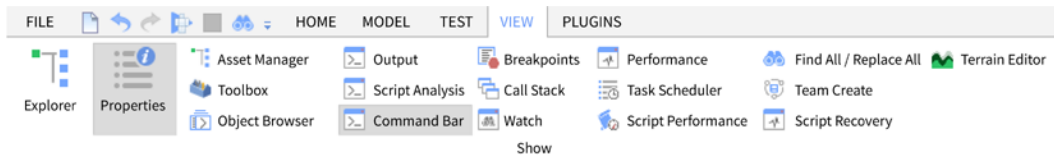


The **World** tab of the menu allows you to change various settings regarding how a visitor interacts with the **Workspace**, specifically in relation to physics and the different properties of a user's character. While this tab may be convenient for some changes, all of the settings here can be done through scripts or by changing the properties of different **services**, which will be explained once you begin programming. An example of changing these settings yourself would be to change the properties of a character's **Humanoid** instance or to simply change the **Gravity** property of the **Workspace** to allow users to jump higher. As a programmer, you should ensure that you know how to make these modifications without the menu, but for quick adjustments, it ultimately comes down to a matter of preference and convenience.



## The View tab

The **View** tab of Studio presents you with many additional tools as well as toggles for different information that can be displayed when developing. Some of these options are shown in *Figure 2.15*. Like the other concepts introduced in this chapter, you should familiarize yourself with these resources so that you are knowledgeable about them as you develop your skillset:



*Figure 2.15: The View tab allows developers to use additional tools and see hidden information*

One of the most crucial Studio tools for managing uploaded assets is the **Asset Manager** window. From this window, you can not only view all the assets that have been uploaded directly to the experience, but you can also use the **Bulk Import** feature. The advantage of using the **Bulk Import** feature is that you can upload up to 50 assets at once to your experience, as opposed to individually uploading them to the website.

The **Team Create** feature will bring up a side window within Studio, a core element of collaboration on projects. **Team Create** allows a project to be edited simultaneously by multiple developers, removing the issue of overwriting the changes made by other developers in different Studio sessions. In order to grant other users permission to edit your experience, you will need to navigate to the **Game Settings** page and navigate to the **Permissions** tab. From there, you can look up users by name and ensure that the **Edit** option is selected for them.

While this will be covered more extensively once you begin learning how to program in Luau, the command bar serves as a spot for directly executing Luau code and can be used when testing or simply in Studio. The **Command Bar** will come to be a precious tool for completing tasks that would be quite time-consuming, such as managing hundreds of assets or manipulating data.

The **Output** window is almost always used alongside the command bar and while conducting tests, as all output and executed code from the command bar is displayed there. More importantly, when a script in your experience produces an error or any other output, it will be directed to this window. The errors, warnings, or prints will also be clickable in this window and bring you to the script and line that produced the output for easy debugging. If Studio needs to display additional information, such as the status of an action or that a user in **Team Create** completed a major action, that information will also be visible in the **Output** menu.

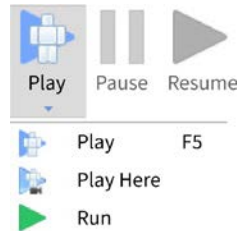
Much like pressing *Shift + F5* shows the current framerate when in an experience (not in Studio) and additional rendering information, toggling the **Summary** option under the **View** tab shows you the framerate while in Studio as well as the rendering information that is normally available while playing an experience.

While Roblox Studio does provide you with many exciting features and tools, the program may freeze or crash when manipulating too many objects at once. However, when this occurs, Roblox virtually always locally saves a file of the experience on your computer. Furthermore, if you were only to crash in a **Team Create** environment, for example, you could use the **Script Recovery** tab to retrieve your lost progress and apply it to the current version of the experience. Let's now look at the tools available for testing your experience.

## The Test tab

The **Test** tab of Studio presents you with several different modes with which to test the functionality of your experience. Before releasing an experience, you should ensure that there are no errors or unwanted behaviors.

In addition to making sure all of the systems of your experience are working, make sure that your experience functions correctly across the desired platforms you want it to be available on. The more platforms you have enabled for your experience, the more potential visitors will be able to engage with your work. To start testing, there are a few initialization modes that can be seen by clicking the drop-down menu below the **Play** button, as depicted in *Figure 2.16*:



*Figure 2.16: There are three different initialization modes for testing*

Clicking **Play** has a couple of behaviors based on how the **Workspace** is configured. If there are **Team** instances parented to the **Team** section of the **Explorer** and a corresponding **Spawn Location** instance in the **Workspace**, then the visitor will spawn randomly at one of the team's spawn locations. If there are no team spawn locations or no teams, the visitor will spawn at a neutral spawn location, if one exists. If there are none, then visitors will spawn around the origin of the **Workspace**, that being position  $(0, 0, 0)$ .

The **Play Here** option spawns you where your camera is positioned regardless of whether spawn locations or teams are present. If you have a script that manually sets the position of a character, this will have precedence as the testing mode positions visitors before a client loads in.

Both the **Play** and **Play Here** options load you in as a client, meaning both local and server scripts will load, and you will have a command bar that can only make local changes. However, by clicking the **Current** button, which is under the **Test** tab, as seen in *Figure 2.17*, you can switch between client and server views, allowing you to make use of the server command bar and also a freely moveable camera. As an additional note, by pressing **Shift + P** when in client view or in an experience, you can switch between the standard and free cameras:



*Figure 2.17: The Pause, Resume, Stop, and Current buttons are only available when testing*

The **Run** option will only run your experience on the server. No clients will be created, and you will only have access to the server command bar and see the **Workspace** strictly from the server view. You will not be able to use the **Current** view feature when in this mode.

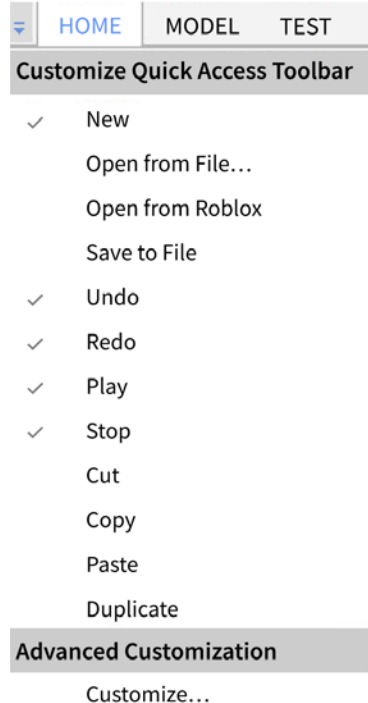
Running a **Local Server** is a way of testing your experience with multiple test users if you do not have someone to test with. Each client (user) you create has a window where you can control it individually and with each test, there is one window that grants you a view from the perspective of the server. You can change how many clients are created using the drop-down menu below the **Test type** button; the range for this option is 0 to 8 clients. We will cover more about the client-server relationship in *Chapter 3, Introduction to Luau*.

The **Team Test** option lets you and those with **Team Create** access test together in an environment much like that of a local server. Some of the benefits of doing a team test, in comparison to running a **Local Server**, include the fact that you only need to control your own user, character appearances load in, and you can use any **Application Programming Interfaces (APIs)** that require the properties of a real user, such as a **UserId** (a **UserId** is a unique identifying number that each Roblox profile has). For clarification, an API is a way for different programs to communicate with each other. If you wanted to get information about your character's appearance, you would use APIs that communicate between your game and the Roblox website. Once you start a **Team Test**, the **Start** icon will change to say **Join** and additionally, a red shutdown button will appear for when you wish to conclude the session.

The tools within the **Emulation** subtab allow you to see how your experience would appear to visitors with different devices and from different regions. This tool is most useful for seeing whether the UI of your experience scales universally across displays of different sizes, how the controls of the experience work when testing with emulation set to a different platform, as well as allowing you to check whether localization is functioning based on a visitor's region. The first tool is the device emulator. Aside from altering your viewport in Studio while editing, by initializing a test, the control scheme will be based on the selected device's platform, meaning you can evaluate mobile, Xbox, and computer interactions while playing. The second tool is an emulator for a user of your own design. From here, you can manipulate their region and alter what policies are in place for that user so that you can make accommodations as needed. In the following section, we will talk about customizing Roblox Studio so you can work more efficiently.

## Customizing Studio to aid your workflow

In Roblox Studio, the tab system is a convenient and customizable system of organization for some of the tools that are available to you. Once you gain experience as a developer, it is natural that you may desire a layout of the tools you use the most to better fit your workflow. By clicking the down-facing arrow next to the Studio tabs, as depicted in *Figure 2.18*, you will be able to add or remove certain actions from the **Quick Access Toolbar**:



*Figure 2.18: This is the customization list for the Quick Access Toolbar*

You will be presented first with a list of basic actions to add or remove to your Toolbar, but by selecting the **Customize...** option, a small dialog box will open and show virtually every button-bound action you can do in Studio with the click of your mouse. Adding or replacing the default action buttons may improve your experience when working in Studio by increasing your overall development efficiency as you will no longer be forced to navigate through multiple pages, in some cases.

Just as customizing your **Quick Access Toolbar** may increase your efficiency in development, there are also numerous add-ons available to make some tedious or otherwise difficult tasks doable with the click of your mouse.

By navigating to the **Toolbox** or **Creator Marketplace** page, you can click the **Plugins** tab and explore a wide variety of plugins that make less-visible information accessible and some seemingly impossible actions doable. For example, scaling something in the **Workspace** by an exact percentage might be difficult but is easy to do with a plugin. As mentioned in the *Creator Marketplace and Avatar Shop* section, you should be cautious of what plugins you install, as you would when adding any asset to your experience. Plugins have the ability to execute programs at the highest level of Roblox security in your project and can alter virtually anything in Studio. This is not said to discourage you from using plugins; as you become more advanced, they may become necessary and there are many developers that dedicate themselves to making quality plugin tools. Before installing a plugin, simply make sure that it is widely used and has a good rating.

With what you have learned from this section, you can begin manipulating different instances that you have added to the **Workspace**, test that your systems function correctly across multiple platforms for different visitors, and personalize Studio in order to most effectively develop your projects. These skills will be particularly important in the following chapter as you begin using Studio to make your first programs. To be fully well-rounded before you begin programming, it would be beneficial to make use of the documentation and other assets that Roblox offers in order to become more knowledgeable about the development process in general, as well as programming constructs.

## Taking advantage of Roblox's resources

Another aspect of the Roblox platform that makes it unique is the abundance of documentation, tutorials, and other resources that are offered to its developers. When we cover more aspects of programming, you will find that the extensive documentation that lists nearly every aspect of what you are working with will be of immense value.

### Tutorials and resources

To conveniently organize information on Luau and other development types, Roblox maintains a separate part of the website for articles and API references. Here, on the developer website, you can learn what functions and events exist for various types of instances and the arguments that each function takes, in addition to logs of Studio updates and documentation on other content relevant to programming.

The website also grants the added benefit of tutorials on several core experience systems for beginners. While this book will teach you how to program in Luau starting from the simplest of concepts, the additional practice and provided experience templates on the developer website are useful for applying what you see to your own projects.

These resources do not require any login or other additional steps to access and can be found here: <https://developer.roblox.com/>.

## The Developer Forum and Talent Hub

The **Developer Forum**, more commonly called the **Dev Forum**, is a section of the Roblox website exclusively dedicated to the promotion of developer discussion, collaboration, and quick-response development help. With multiple categories that are devoted to guidance on separate fields of development, most of the obscure issues you may run into when developing a project may be resolved within a matter of minutes, if they haven't already been previously resolved by someone with the same complication in the past.

Before you can post new threads on the forum, you must complete a tutorial and spend a certain amount of time on the website. This is to prevent spam and unwanted content from flooding the various channels. If you build up a reputation on the Dev Forum, earning solutions and likes awarded by other users, you may gain additional permissions and access to a few additional channels. To get started, you can find the Dev Forum here: <https://devforum.roblox.com/>.

The **Talent Hub** is a relatively new resource that Roblox has created so developers can have a dedicated space to show their portfolios and look for work or hire other developers. From the perspective of Roblox, there is a unique culture for collaboration on the platform that they wish to nurture and this is another way to do that.

In order to participate, there are different criteria that must be met depending on who you are. For example, in order to post in certain sections or be eligible for different offers, you must be a verified user. You can become a verified user by verifying your age on Roblox, which is necessary for some features like voice chat, or by having gone through the DevEx program, you are automatically considered a verified user on the Talent Hub. To learn more about the Talent Hub and its requirements you can find the announcement thread here: <https://devforum.roblox.com/t/introducing-talent-hub-open-beta-new-platform-to-find-post-work/1396502>.

As mentioned in the previous chapter, staying social and making new connections with other developers is the best thing you can do to advance yourself as a developer and increase your level of recognition on the platform. While the Dev Forum and Talent Hub are not considered formal, maintaining a semi-professional demeanor may help you find promotion and work quicker. By utilizing both the Dev Forum, Talent Hub, and the Developer website, you will not only gain more confidence as a developer, but you will also be promoting yourself in the community as you help others or receive help yourself.

## Summary

In this chapter, you learned how to use the **Create** page of the Roblox website in order to create new experiences, change external settings, add monetization items, and promote your experience for millions of users to see. Moreover, you became familiar with Roblox Studio, and you can now create and change the properties of new instances, manage internal game settings, customize Studio to increase your productivity, and manipulate your environment with built-in tools. Lastly, you found the resources that Roblox provides to its developers to advance knowledge, get development help, and network with other members of the community.

The next chapter will begin by introducing you to the Luau language and general programming constructs that will be key components of your future as a game developer and programmer in general. By the end of the following chapter, you will be able to make your own programs in Luau with the prowess of utilizing the proper style to produce optimized code.

## Worksheet

- Q1. What page should you visit if you want to start developing?
- Q2. If you want to make your experience public or private, which menu should you visit?
- Q3. If you want to change the appearance of your game, such as the icon, thumbnail, or description, which menu should you visit?
- Q4. If you want to change what platforms players can play your game on, or how many players can fit in a server, which menu should you visit?
- Q5. When your game is ready to release and you want to promote it, which menu should you go to?
- Q6. If you want your work in Roblox Studio to be playable from the Roblox website, what must you do?
- Q7. What is the word that refers to any kind of object in Roblox?
- Q8. Roblox uses what to change the behaviors and appearances of different instances?
- Q9. What are the five tools to physically manipulate objects in Roblox Studio?
- Q10. Where can you quickly change many game-related settings from Roblox Studio?
- Q11. How can you load in and test your experience in Roblox Studio?
- Q12. What is the name of the forum where you can participate in developer discussions and get help and feedback from the Roblox developer community?





---

# Section 2

---

## Programming in Roblox

This part will cover the most comprehensive section of the book: programming Roblox games. You will learn how to program in Luau from the ground up; no prior programming experience will be assumed. We will then apply the topics we have learned throughout to create full experiences at the end.

This section comprises the following chapters:

- *Chapter 3, Introduction to Luau*
- *Chapter 4, Roblox Programming Scenarios*
- *Chapter 5, Creating an Obby*
- *Chapter 6, Creating a Battle Royale Game*



The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy-2nd-Edition>.



# 3

## Introduction to Luau

The **Luau** language is a fast, procedural programming language adapted from **Lua**. Lua was originally created in 1993 due to software trade barriers in Brazil, which prevented many from buying specialized software from outside the country. Because of this, the language was designed to be highly customizable and C-based so that programmers could easily make changes to fit their needs. When Roblox began, they used Lua 5.1 but as the platform has expanded, they've made enough additions and changes to warrant their own scripting language, Luau. It should be noted that all Lua 5.1 code is valid in Luau, as it is a subset of the current language. That is to say, syntactically, everything is the same but new features have been added.

The goal of this chapter is to give you the knowledge you need to make your first programs in Luau so that you're ready to become a fully proficient programmer. No prior programming experience is assumed, so we will start by covering the concept of variables and other universal programming constructs. Then, within a few chapters, you will be able to create full experiences, of your own design, on Roblox. If you find some material difficult, don't be worried! You're taking a crash course in a skill that you will build upon for the rest of your life and this is just the beginning. If you feel the need to move between different sections or review some earlier ones, you are encouraged to do so!

In this chapter, we're going to cover the following main topics:

- Learning about data types and creating variables
- Declaring and using loops
- Learning about functions and events
- Demonstrating programming style and efficiency

Let's get started!

## Technical requirements

In this chapter, you will be working entirely in Studio and must meet the requirements mentioned in the *Technical requirements* section of *Chapter 2, Knowing Your Work Environment*. As a note, you will never need to check for updates on either Roblox Studio or **Roblox Player** as they will update automatically when you start them. This will ensure that any new features and security measures have been enabled.

You can find all the code used in this chapter in the book's GitHub repository at <https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy-2nd-Edition/tree/main/Chapter03>.

## Learning about data types and creating variables

In programming, a **variable** is a way for your code to hold a piece of information; **different types of information are called data types**. Variables are convenient because when you create one, you can give it an identifier (a name) so that it may be easily referenced and used later. In many programming languages, these variables are **typed**; this means that the type of the variable, be it a number or a word, must be decided when it is created.

Luau is a **gradually typed** language; this means that variables are not typed by default, but you can specify them to be so. For this book, we will not be making use of gradual typing, which was the only option available before the implementation of Luau in Roblox and is more widely used. You can learn more about gradual typing and other features of the Luau language here: <https://luau-lang.org/>.

It is important that you know what the most common data types are before you begin programming.

## Data types

In Luau, the **number** data type can hold both whole and fractional portions of numeric values. Examples of a number can include 1, 0.33, -1.2, etc. This is the only numeric type used in Lua and Luau but it may be helpful to remember that an integer is simply a whole number, positive, negative, or zero. The number data type is actually named a **double-precision floating-point number**, though it is more commonly called a **double**, which is a type of float. Some values cannot be perfectly represented by this system because of how computers work; such inaccuracies are referred to as **floating-point errors**. It's important to be aware of these errors when processing data and you may run into them in a game development environment.

**Booleans** are a simple data type with a binary true or false value; when stored in a variable, they are more frequently called **bools**. It is important to note that in Luau, 1 and 0 only have numeric values and do not have any uses in Boolean logic, as seen in other languages. That is to say, 1 and 0 only represent numbers, not true or false or any other value.

**Strings**, as a data type, are best explained as any kind of text. Strings can represent words, sentences, letters, and more. In most languages, strings are an array or sequence of individual characters; in Luau, however, characters are not a data type.

In Luau, **tables** are a data structure that can be used in many different ways. We will discuss two uses of tables in this section: **arrays** and **dictionaries**. Unlike arrays in other programming languages, tables when used as an array can act more like lists, as they are not limited to an initialized size, nor do they require one; as a result, additional table positions do not need to be preemptively reserved. Elements contained in these tables are indexed from 1 to **n**, with **n** being how many elements are in your table. It should be noted that in most other languages, the starting index value of a table is 0 whereas it is 1 in Luau; these are called zero-based and one-based indexing styles respectively. Additionally, tables are not typed by default; it could be said that this has many advantages, as in other languages you would be restricted to only adding values of the same type into a single array. However, you should still keep your tables organized and not loosely throw any type of data into them like a virtual colander.

The greatest benefit of using a table as a **dictionary** is the ability to index anything with a convenient key instead of numbered indices. For example, if you had a farm in your experience and you wanted all the apples in your experience to have one functionality and all bananas to have another, you could use the name of the fruit as a dictionary key that has an associated function; you could even use the physical fruit itself as the key since keys are not limited just to strings. As before, dictionaries have an infinite capacity and elements within them are still not typed by default. In mathematics, a vector is a quantity that contains both direction and magnitude. These can be used to represent data about position, rotation, velocity, and more. In Luau, vectors are a custom data type made by Roblox, not a Luau **primitive** (native data type), like those previously mentioned. This means that Roblox can design data types like vectors to have additional properties and methods for different uses.

There are two specific vector types you will commonly work with: the **Vector3** and **Vector2** data types. A **Vector3** contains *X*, *Y*, and *Z* components and is used to define the position and orientation of instances such as parts. A **Vector2** is typically only used when working with a UI or in any other two-dimensional scenario; it contains only *X* and *Y* components.

Vectors are useful for performing various calculations, from checking the distance between two positions to computations such as cross and dot products. For example, if you wanted to display a prompt to a user once they are within a certain distance, you would need to use the positions of both the user and the origin of the prompt (both of which are vectors) and calculate the distance (a scalar):

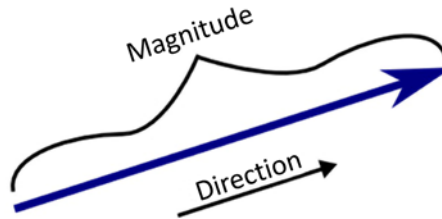


Figure 3.1: A vector is capable of conveying information about both direction and magnitude

A **CFrame**, or coordinate frame, is another type of userdata similar to that of a vector but has the ability to hold additional information. Similarly, CFrames contain positional data but also include nine elements that make up a  $3 \times 3$  matrix describing the rotation of the CFrame. Because of this, most CFrame manipulations allow for positional changes to be relative to the orientation of the coordinate frame itself. For clarification, to manipulate a data type is simply to change its value by way of some operation. For example, adding 1 to a variable that holds the value 4 to make it have a value of 5.

**Instances** are a userdata and represent everything that you can interact with in the **Explorer** window of Studio. Different types of instances are called **classes**, and each class possesses different associated properties. To see the extent of instance classes, view the full list on the developer website: <https://developer.roblox.com/en-us/api-reference/index>.



While you have learned about all the primitive data types Luau contains, there are many more userdata types Roblox has that would not be relevant to mention in this section. The full list of Roblox userdata types can be found here: <https://developer.roblox.com/en-us/api-reference/data-types>.

Next, you will learn how to handle these different data types by assigning them to variables. **Variables** allow you to hold and manipulate data so that it can be used with any program that you write.

## Setting and manipulating variables

Initializing and changing variables in Luau is a process that utilizes a variety of different **operators** to achieve a desired result. A convenient way to check the value of your variable is by using the `print()` function. For many programmers, `print("Hello World!")` is the first line of code they ever write. The `print()` function is a valuable tool when following along in your program and for observing what your code produces when it would otherwise not be visible. In this section, we will begin writing our first lines of code. You should open a place in Studio and either use the **Command Bar** and **Output Window** or insert a **Script** into your experience via the **Explorer** window. If you are unfamiliar with how to access these windows, revisit the *View Tab* part of the *Getting Started with Roblox Studio* section of *Chapter 2, Knowing Your Work Environment*:

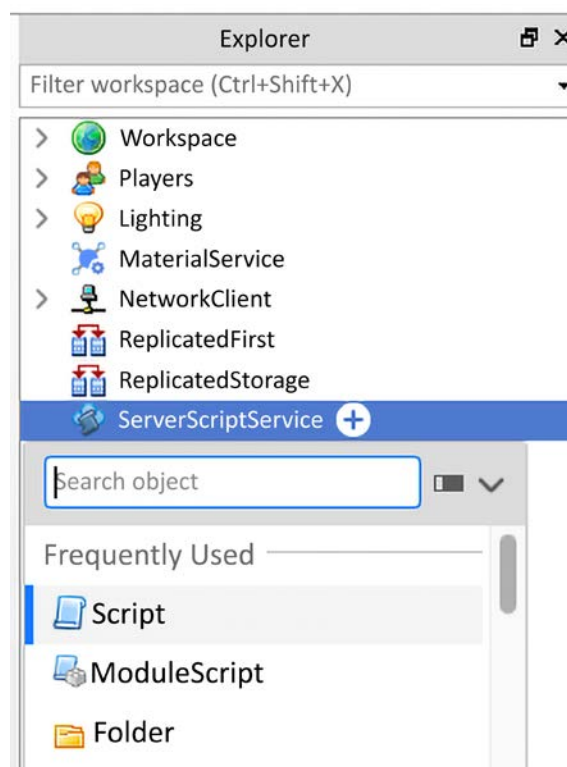


Figure 3.2: You can insert objects such as a *Script* by hovering over something in the Explorer window and pressing the +



## Numbers

**Number variables** are easy to set and change. If you want to initialize a variable, you should utilize the `local` keyword. Initializing a variable without this keyword will work, but it defines the variable for the entire script once it has been run, which is unnecessary in almost all cases and is considered poor style. After writing `local`, you put the name of your variable. A variable name cannot start with non-alphabetical characters and cannot contain any characters that are not alphanumeric except for underscores. For example, if you wanted to have a variable that simply held the value of 99, your code would look like this:

```
local myNumber = 99
```

There are many different operators we can use to change the variable, and **libraries** of special functions, but for this example, we simply want the value to increment by 1 to reach a total value of 100. To accomplish this, we can think of setting the variable to itself plus 1 by using the addition operator (+):

```
myNumber = myNumber + 1
```

A functionality offered by Luau is **compound operators**. Compound operators are a convenient way of modifying data as less needs to be written by the programmer (you!).

So, going forward, instead of writing that a variable should be equal to itself plus 1 to increment it, you can do the following, which means the exact same:

```
myNumber += 1
```

You may have noticed that `local` does not precede the variable name here. This is because `local` is only put before your variable name when you're initializing it; to reference or alter a variable, you simply write the variable's name. Depending on the scenario, it may be more practical to simply set the variable to 100 directly. In this case, you would simply set the variable to the value, similar to what you did when initializing it (without the `local` statement, of course).

Luau supports the arithmetic operators that are standard across most languages, those being for addition (+), subtraction (-), multiplication (\*), division (/), and modulo (%). Like the previous example, all of these can be expressed as compound operators when followed by an equals sign (=). For more advanced operations, Luau provides a library with the same functionality as the standard `math` library in the C language. This library provides trigonometric functions, value conversions, and specific values with convenience and accuracy. To utilize this library, we can use the `math` keyword.

Here is an example of getting an accurate approximation of pi by using the math library:

```
myNumber = math.pi
```

Moving forward, we will discuss the **Boolean** data type.

## Booleans

Setting a **Boolean** is simple as there are only two initialization options: using the true or false keyword. An initialization statement for a Boolean value may look something like this:

```
local myBool = true
```

To change the value of this variable, you simply need to set the bool as true or false. For example, we will set myBool equal to false with the following line of code:

```
myBool = false
```

There is a trick for setting a bool to its opposite value in one line, as opposed to using a conditional. We can do this by making use of the not operator, which will be covered more once we get to conditional statements. The not operator serves to simply return the opposite of the input following it. For example, if we wanted to change the preceding myBool variable from false back to true, we could simply use the following line of code:

```
myBool = not myBool  
print(myBool) -> true
```

Continuing, we will cover another primitive data type, **strings**.

## Strings

To declare a **string**, you should use the same variable initialization style and write your text inside double quotes. While single quotes can be used, double quotes are the stylistic choice, unless double quotes are contained within the string itself:

```
local myString = "Hello"
```

If your string contains double quotes, Lua uses the backslash (\) as an escape character. This means that any character that would normally be special is treated as text within a string. For example, if someone in some game dialog is speaking from the third person, you could create double quote marks, like this:

```
myString = "He said \"I don't like apples!\""
```

Conversely, this backslash operator makes some normal text special. Two characters that are made special by the backslash character are the letters `n` and `t`. When a `\t` is present within your string, a tab will be added in that place; as a side note, a tab is considered a single character by computer systems. When a `\n` is in your string, a new line is inserted at that point, for example:

```
myString = "Separated\tby\ttabs"
print(myString) -> "Separated    by    tabs"
myString = "Separated\nby\nlines"
print(myString) ->
"Separated
by
lines"
```

If you have multiple lines in your string, you do not necessarily need to utilize the `\n` operator. Lua, unlike some other languages, supports the use of multiline strings. Aside from being able to simply press your *Enter* key to create new lines, you can more conveniently format paragraph-sized strings in your programs. To initialize a paragraph string, you must capture your text within double brackets, as shown here:

```
myString = [[This string
can span
multiple lines.]]
```

One of the most common ways string variables can be changed is by **concatenating** them. By following any string with `..` and providing another string, the latter string will be attached at that position:

```
myString = "Hello"
myString ..= " World!"
print(myString) -> "Hello World!"
```

The ability to concatenate is particularly useful when you're presenting different information to a player via a UI element. For example, if you wanted to announce who the winner of the round of a game is, you can append the name of that player to a string:

```
local winnerName = "WinnerWinner"
myString = "Game over! " .. winnerName .. " has won the round!"
print(myString) -> "Game over! WinnerWinner has won the round!"
```

Similar to how numeric data has a library for mathematical operations, there exists a library of string functions for more complex manipulations, as well as data management. This library can be accessed by writing the `string` keyword. Some functions include the ability to change the case of all letters within a string, the ability to split strings at certain points, and even to find all strings within a string that match a certain pattern, which is useful for systems such as in-game search bars. For example, all the letters in the following string will be converted into uppercase using one of the **string** library's functions:

```
myString = "iT iS wARm tOdAY."  
print(string.upper(myString)) -> "IT IS WARM TODAY."
```

Using strings in numeric arithmetic should be avoided when possible, but there may be situations where this happens. Whenever a string is used where a number is required, Luau will attempt to automatically convert that string into a number. For example, if you try to add the string "50" to the number 100, it will function correctly, as shown here:

```
print("50" + 100) -> 150
```

However, if the string you are attempting to perform an operation on contains non-numeric characters, the string-to-number conversion will fail. To prevent this, you can check if a string is fully numeric by using the `tonumber()` function. If the string that's been passed to the function cannot be converted into a number, the value that's returned will be `nil`; `nil` is a value that represents something non-existent. If we attempt to add the string "Hello" to the number 100, an error will occur:

```
myString = "Hello"  
print(tonumber(myString)) -> nil  
local myNumber = 100 + myString -> "local myNumber = 100 + myString:3:  
attempt to perform arithmetic (add) on number and string"
```

Next, you will learn about **tables**, which have a wide variety of applications.

## Tables

**Tables** are straightforward but less intuitive to set and manipulate than the other data types we have covered so far, as you must make use of a library for some operations. To create a new, empty table, you must set your variable to a set of braces (`{}`), as shown here:

```
local myTable = {}
```

When initializing a new table, you do not need to have it start out empty; you can include elements within your table when it is first created. It is important to note that elements in tables require a separating character; in this case, that character can be either a comma (,) or semicolon (;). For example, if a player was tasked with retrieving items from a grocery list, you could initialize a table of predetermined foodstuffs this way:

```
local myTable = {"Tofu", "Milk", "Bacon"}
```

Once you've created your table, you will need to be able to index what items exist within your list. Without loops, which will be covered later in this chapter, you can only index items individually. Remember that tables use one-based numeric indexing, so indexing items is just done with a number within brackets ([ ]). All items from the grocery list could be either assigned to a variable or acquired directly, as seen in the following print line of code:

```
local myTable = {"Tofu", "Milk", "Bacon"}
local firstItem = myTable[1]
print(firstItem, myTable[2], myTable[3]) -> "Tofu Milk Bacon"
```

To add or remove elements from a table, you can use the `table` library, which can be accessed by using the `table` keyword. This library allows you to alter the structure of tables by changing how they are sorted, what their contents are, and where existing table entries are located. In order to add new elements to a table, you should use the `table.insert()` function. The function requires a minimum of two arguments, with the first being the table being targeted and the second being the value that you wish to add to the table. If three arguments are provided, the first argument is the targeted table, the second is the desired table position, and the third is the value to be added. When using the function with three arguments, it is important to remember that all the elements following or at the desired position are shifted to the right. Furthermore, there are no restrictions to the provided index, meaning the index can be negative or be an element that hasn't been reached yet by the length of the table, though you should avoid this. Here is an example of adding an element to the beginning of a table and an element without a position specified, which will, by default, go to the end of the table:

```
local items = {"Elephant", "Towel", "Turtle"}
table.insert(items, 1, "Rock")
table.insert(items, "Cat")
-> items = {"Rock", "Elephant", "Towel", "Turtle", "Cat"}
```

Without loops, you cannot remove all the elements of a specified value or those that meet some criteria with complete certainty.

In this case, you will need to know the index of the value you want to be removed from the table. For example, if the list is only supposed to contain living things, we would want to remove the Rock and Towel items. We can do this by using the `table.remove()` function. It is important to note that removing an element from a table will shift all the elements that follow it to the left. So, if the rock was removed from the table first, the indices of all the other items in the table would be one less than they were before. This can be seen in the following code:

```
items = {"Rock", "Elephant", "Towel", "Turtle", "Cat"}
table.remove(items, 1)
-> items = {"Elephant", "Towel", "Turtle", "Cat"}
table.remove(items, 2)
-> items = {"Elephant", "Turtle", "Cat"}
```

To confirm that the correct number of elements are within your table at any given time, you can preface a table or table variable with the `#` operator to return the number of elements within it. Additionally, you can use the `table.getn()` function to return the same result, though this is longer to write. You can prove these techniques return the same result by making the following comparison:

```
print(#items == table.getn(items)) -> true
```

We'll cover how to make more comparisons when we learn about conditional statements. In the following section, you will learn about **dictionaries**.

## Dictionaries

As we mentioned previously, **dictionaries** are tables that use custom, key-based indexing as opposed to sorted numeric indexes. Conceptually, you can think of entering values into a dictionary as declaring a variable, except the `local` keyword is not applicable here. While elements in a dictionary can be laid out like a table, it is more common for each entry to have its own line; the separating character for elements can be either a semi-colon or a comma. If you had a restaurant's menu within your experience, you could arrange the items within a dictionary, with the key being the name of the meal's course and the value being the name of the dish:

```
local menu = {
  appetizer = "House salad";
  entree = "Ham sandwich";
  dessert = "Ice cream";
}
```

Indexing these declared elements is quite intuitive as you must simply follow the path to the desired value. In this case, let's say you wanted to capture what dish was being served as the entrée on the menu with a new variable:

```
local meal = menu.entree
print(meal) -> "Ham sandwich"
```

Setting elements is equally as straightforward: by following the path, you can set or alter the element based on its data type like any other variable:

```
menu.entree = "Turkey sandwich"
```

One of the advantages of using these keys in Luau is that they are not restricted to only string indexes. By using brackets ([ ]), you can use any data type as an index of your value. This is particularly useful if you want one data type to have a direct association with another at a given value. For example, if you wanted to set a list of threshold prices that correlated with a describing string, you could use a number as an index. Bear in mind that in order to index non-string keys, you must also use brackets:

```
local prices = {
    [0] = "Free";
    [5] = "Cheap";
    [20] = "Average";
    [50] = "Expensive";
}
print(prices[0]) -> "Free"
```

What may be less obvious is the ability to use userdata values as keys. We could associate the origin of our **Workspace** with a string, number, or another position; you are not restricted in any regard.

Something to note is that tables can have another table as a value; whenever something exists within another entity of the same type, we call this **nesting**. You can create structures by nesting tables within each other and fetching them with the same key-based style. When we discuss modules, in addition to other script types, you'll see that nesting tables is a somewhat common practice for organizational and functional purposes. For instance, if you wanted to list some basic stats of **non-playable characters (NPCs)** in your experiences, it may make sense to include those stats in a table under one collective NPC table so that the information can be indexed by the name of the NPC:

```
local units = {  
    ["Heavy Soldier"] = {  
        WalkSpeed = 16;  
        Damage = 25;  
    };  
    Scout = {  
        WalkSpeed = 25;  
        Damage = 15;  
    };  
}
```

You will now learn about **vectors**, which will allow you to better understand 3D environments.

## Vectors

**Vectors** are values that represent both direction and magnitude. In Roblox programming, vectors are used to represent the positions in three- and two-dimensional environments, define the orientations of different instances, show the direction of CFrames, and calculate additional information about objects in relation to each other.

Declaring a vector is much like creating many other Roblox userdata types. After stating the name of the userdata, you choose the constructing function. In most cases, when you're working with vectors, you will use the new option. For this example, we will be using a Vector3, though a Vector2 follows the same format but will have only two components:

```
local myVector = Vector3.new(0,0,0) --Vector3.new() will also produce a  
0-vector
```

Changing vector values is a little different from changing the other data types we have covered. This is because arithmetic is done across all components with changing behaviors, depending on what is being used in the operation. To demonstrate this, arithmetic between two vectors is done by component, meaning that adding two vectors will combine the values of each component; you can conceptualize this as the vectors being physically over each other and adding each column together:

```
myVector = Vector3.new(1,3,5) + Vector3.new(2,4,6)  
-> Vector3.new(3,7,11)
```



However, the behavior of vector arithmetic changes when **scalar** values are present. A **scalar** is any value that conveys magnitude but not direction. For example, vectors can be both multiplied and divided by scalars, but you cannot perform addition or subtraction with these mismatched data types. The only exception to this is when a scalar is divided by a vector, in which case division is done by component, with the scalar acting as the numerator (the number being divided) of each element:

```
myVector = Vector3.new(2,4,6) * 2 -> Vector3.new(4,8,12)
myVector = Vector3.new(2,4,6) / 2 -> Vector3.new(1,2,3)
myVector = 2 / Vector3.new(2,4,6) -> Vector3.new(1,0.5,0.333)
```

Aside from changing vectors as a whole, you can capture individual values from a vector. Here, we are setting three local variables at once. Typically, this format is reserved for what is called a **tuple**; a tuple is essentially when a function returns multiple values that are not grouped together in a structure such as a table and, consequently, more than one variable must be assigned to one statement. By indexing the X, Y, and Z fields of the vector, we can capture number values, which can be used in various computations:

```
local x,y,z = myVector.X, myVector.Y, myVector.Z
```

One of the most common calculations that's done with vectors is finding the distance between two positions. While you could use the distance formula using the `math` library, there is a more direct way to do this. As we mentioned previously, all vectors have magnitude; this magnitude can also be manually calculated. However, Roblox includes a `magnitude` property for all vectors that can be captured, as shown here:

```
local magnitude = myVector.Magnitude
```

To calculate the distance between two positional vectors, the vectors must be subtracted from each other; then, the resulting magnitude of the new vector will be the distance between them:

```
local vector1 = Vector3.new(1,5,7)
local vector2 = Vector3.new(2,4,6)
local distance = (vector1 - vector2).Magnitude
print(distance) -> 1.73205
```

Like vectors, `CFrames` are a data type used in 3D environments. We will discuss them in the next section.

## CFrames

The userdata **CFrame** is similar to a vector but has a wider range of uses and behaviors because of the additional information it carries. Declaring a CFrame variable with only positional data is the same as what you'd do with a vector; you can use the new constructor and must provide X, Y, and Z coordinates, as shown here:

```
local myCFrame = CFrame.new(0,0,0) --CFrame.new() can also be used here.
```

What makes a CFrame distinct from a vector is its matrix of rotational information, which describes its orientation via directional vectors. While you will likely not deal with these matrix components individually, changing the orientation of a CFrame, is something you will be doing constantly. A typical way to set the orientation of a CFrame is to use the `lookAt` CFrame constructor. By providing an origin and target, a new CFrame will be created at the provided position, with `LookVector` (the front of the part being manipulated) directed toward the specified `lookAt` position. The greatest advantage of this is for making parts look at a different position for ease of movement relative to its forward-facing direction. To test this, add two parts to our **Workspace** named `Part1` and `Part2`. Position `Part1` wherever you would like and place `Part2` where you want the first part to face toward. Once you've done this, execute the following code to see how the front face of `Part1` now points directly at `Part2`:

```
local Part1 = workspace.Part1
local Part2 = workspace.Part2
Part1.CFrame = CFrame.lookAt(Part1.Position, Part2.Position)
```

As we mentioned previously, one of the benefits of manipulating CFrame rotation is for relative movement. If you've already played around with the `position` property of our parts, you may have noticed that the position is global and there is not really a way for a part to move in the direction it is facing. This global view is called the **world space**. By using CFrame arithmetic, you can move parts in relation to their orientation, which is important for systems such as projectiles, doors, and even vehicles; this relative view is called the **object space**. Here is a simple application where you could move a part forward in the direction it is facing by one stud:

```
myCFrame *= CFrame.new(0,0,-1)
```

Take note that multiplying two CFrame values does not actually multiply the components, and the operation that's occurring is conceptually more like addition. When the preceding code is implemented inside a loop, particularly a fast-running one, you can simulate movement.

This is a technique that's often used for projectile systems as many resources are saved since the *moving* part is anchored, so physics calculations do not need to be made. This directional movement is used for almost any moving part and in most cases should be used instead of the position property.

While it may not be intuitive, you must be aware of which axes control the rotational behavior you seek in a three-dimensional environment. As shown in the following image, the X, Y, and Z axes may control different directions of rotation than you may have anticipated:

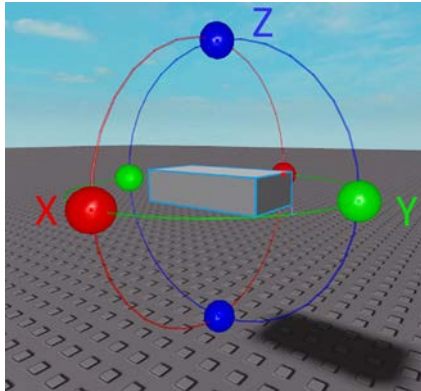


Figure 3.3: Looking at the front of this part, you can see which axes cause what rotational behavior

You may be wondering why the handles for the Y axis are horizontal and the handles for the X axis vertical. The reason behind this is that the axes of rotation are defined by the object moving around the axis. Due to this, they will not stand in the same orientation that the axes lay on a graph. So, in the case of an object being rotated around the Y axis, you would need horizontal handles:

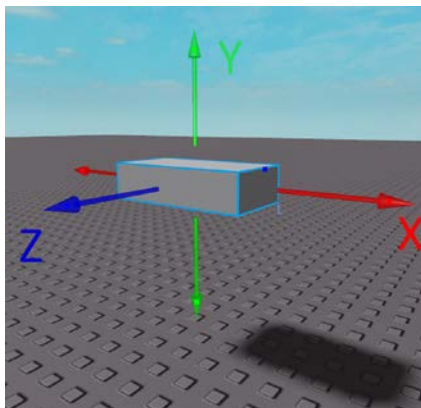


Figure 3.4: Here, you can see that the handles correspond to the object rotating around the axes

By using the `CFrame.Angles()` constructor, you can directly manipulate the orientation of a `CFrame`. Remember that parts use degrees for their `orientation` property, but `CFrames` use radians when working with their rotational matrix. If you are unfamiliar with radians, they are a unit used to measure angles just like degrees. While keeping the same behavior as other `CFrame` arithmetic cases, multiplying a `CFrame` by `CFrame.Angles()` effectively adds to each orientation component.

Let's follow an example where we want a `CFrame` to fully turn around by 180 degrees. To achieve this, we will need to rotate around the *Y* axis and use radians. While we can use `math.pi` (`pi` being the equivalent of 180 degrees in radians) we can convert 180 degrees to radians by way of the `rad()` function of the `math` library. You can see our final code here:

```
myCFrame = CFrame.new() --No rotation
myCFrame *= CFrame.Angles(0,math.pi,0) --you can also use math.rad(180)
for math.pi
```

`CFrames` have many built-in functions but do not use a library like the Luau primitive types we have already covered. There are many functions and constructors – even some redundant ones – that all have their own uses for accomplishing complex operations for use in a variety of world-based computations. While there are many functions you will not likely use many of them unless you're making math-intensive systems. However, say you wanted to obtain the orientation of the `CFrame` we just set. As stated previously, `CFrames` use a matrix of directional vectors to describe their rotation; you are unable to just index this value like a part's orientation. Isolating this orientation for use with parts or calculations is not immediately apparent and requires the use of a special function. To extract this information, we can use the `ToOrientation()` function. This function will return three numbers as a tuple in the normal *X, Y, Z* order. These values represent a close approximation of the rotation of the `CFrame` in radians. Using the `CFrame` value from the previous example, we can capture all three components of the orientation within a `Vector3` without assigning the values to variables:

```
local orientation = Vector3.new(myCFrame:ToOrientation())
-> Vector3.new(-0,3.1415925,0)
```

With this new value, there are several types of applications you may want to perform. This orientation can now be applied to a part in our **Workspace**; this can be done by multiplying the vector by a conversion value in order for the orientation to be in degrees:

```
part.Orientation = orientation * (360 / (2 * math.pi))
```

It should be noted that this is not a very direct process and Roblox recommends that if you want to change the orientation of the part, to simply change the part's `CFrame`, like in the examples above.

If you are interested in exploring more information about other `CFrame` functions and constructors, you can view all of them by visiting the API reference for the topic on the developer website here: <https://developer.roblox.com/en-us/api-reference/datatype/CFrame>.

## Instances

**Instances** are userdata and subsequently are also created by using the new constructor. There are hundreds of different instances, though only some can be created from scripts due to security permissions. Instances will be covered much more in the following chapter, but for now, let's simply make a new part in our **Workspace** and color it a bright cyan. To do this, we can use `Instance.new()` and provide a string of the class name as its argument:

```
local part = Instance.new("Part")
part.BrickColor = BrickColor.new("Cyan")
part.Parent = workspace
```

Now that you know how to set and alter these data types once they have been assigned to variables, you will learn how to check the values of them to determine what type of behavior should occur as a result.

## Conditional statements

**Conditional statements** or **conditional expressions** are used in code when you want different behaviors to occur only when some requirement is met. These are important for determining different information about data and what your program should do to handle that data accordingly.

The `if` statement is the core component of conditional expressions. These statements consist of three elements: the `if` keyword, the case that must be met for the contained code to be executed, and the `then` keyword, which serves as an identifier for the end of your case. To give you a direct example of this, the following code shows a conditional where the condition is simply true, meaning the contained instructions will always be executed:

```
if true then
    print("Executed")
end
```

Here, you can see that the conditional closes with the `end` keyword. In Luau, anything that acts as a single block of code (defines a scope) will have `end` designate the conclusion of that block.

Returning to the condition portion of `if` statements, determining whether a condition has been met is based on Boolean logic, a system of evaluating any type of data. We end up with a `true` or `false` value as the result. To do this, evaluations are made using a system containing various **logical operators** and **relational operators**. Like many languages, Luau uses two equals signs (`==`) to check equality between values; this is a relational operator. As an example, let's say you want to make a part collidable only if its transparency is equal to `1`; for this scenario, `part` is arbitrarily defined, meaning we haven't defined it but this is code you can use with a part:

```
if part.Transparency == 1 then
    part.CanCollide = true
end
```

For this operator, there exists an opposite: the *not equal to* expression (`~=`). Like the use of its counterpart, this relational operator is used to make comparisons of an explicit value.

To check finite or infinite ranges of numbers, you can use the relational operators of greater than (`>`) and less than (`<`). These operators have variations that include the value they are being compared to in the form of greater than or equal to (`>=`) and less than or equal to (`<=`). A very practical application of these operators is simply checking if a player is alive. In this example, assume that `char` is already defined as a player's character:

```
local humanoid = char.Humanoid
if humanoid.Health > 0 then
    print("Player is alive!")
end
```

The `not` operator serves to negate whatever value is provided to it. As we saw previously when we switched the state of a `bool` variable, the `not` operator returned the opposite of what was given to it. In terms of conditional statements, this can be used in similar situations as the `~=` operator:

```
if not part.Anchored then
    part.Material = Enum.Material.Neon
end
```

The `and` operator is used to compare two values and requires that the values provided to it are both `true`. In the following example, we want to ensure that both variables hold `Fruit` as their value.

When this condition is not met because one is defined as `Vegetable`, we will not see any output since a true value is not present on both sides of the operator:

```
local item1 = "Fruit"
local item2 = "Vegetable"
if item1 == "Fruit" and item2 == "Fruit" then
    print("Both fruit.") --No output as requirements not met.
end
```

The logical or operator only requires that at least one of the values it receives is true. We can see in this instance that the item is defined as `Vegetable`. The condition says that the item must be defined as either `Fruit` or `Vegetable`, meaning we will see output since one of the values on either side of the operator is true:

```
local item = "Vegetable"
if item == "Fruit" or item == "Vegetable" then
    print("Is produce.") --Prints as one requirement is met.
end
```

Commonly used in conditional statements in the Roblox environment when working with instances is the `IsA` method. This method allows you to check whether an instance is of a specified class. This differs from the `ClassName` property of an instance as `IsA` allows you to check **base classes**. Base classes exist as groups for some instances that share the same characteristics. For example, `Parts`, `MeshParts`, and `UnionOperations` are all `BaseParts`, a base class. So, if you wanted to ensure whatever instance you are interacting with is a `BasePart` before printing its transparency, you could do this:

```
if myInstance:IsA("BasePart") then
    print(myInstance.Name.. "'s transparency is " .. myInstance.Transparency)
end
```

As we mentioned previously, you can check multiple cases using one conditional expression, though this does not require the use of multiple `if` statements. The `else` keyword grants additional functionality to these expressions by executing an alternate case if the first condition was not passed. Let's look at an example where lifting a heavy object requires 100 strength but an object that is not heavy requires only 50 strength.

Notice that our heavy variable, being a bool, will only be true or false and consequently, we do not need to use the equality operator (==):

```
local heavy = true
local strengthRequired = 0

if heavy then
    strengthRequired = 100
else
    strengthRequired = 50
end

print(strengthRequired) -> 100
```

While this has great uses, it does not allow us to explicitly check additional cases – it merely gives us an idea of what to do if the previous condition was not satisfied. The `elseif` keyword is used when you want to check additional cases that may occur under different conditions. You can have as many `elseif` statements as desired, allowing you to create a chain of various conditions and cases. When using `elseif` statements, you can still utilize an `else` expression, but it must be at the end of the overall conditional statement. Let's look at an example where a machine has been supplied random produce and we must count fruits, vegetables, as well as any other item that managed to find its way into the supply:

```
local numFruits = 0
local numVeggies = 0
local notProduce = 0
local item = "Fruit"

if item == "Fruit" then
    numFruits += 1
elseif item == "Vegetable" then
    numVeggies += 1
else
    notProduce += 1
end
```

Lastly, there exist implicit conditional statements, which are expressions where, through the use of logical operators, you can set a condition and alternate cases without ever explicitly writing an `if` statement. In most languages, this behavior is called a **ternary expression**.



In the following code, the goal is to assign a string to the `isAnchored` variable based on whether the part is, in fact, anchored. While this could be accomplished with an `if-else` statement, it is shorter to use some logic here instead:

```
local isAnchored = Part.Anchored and "Anchored" or "Unanchored"
```

As you can see, if the part is anchored, that side of the `and` operator will be true when assigning "Anchored" to the variable by using **short-circuit logic**, meaning conditional evaluation stops after one condition is met or violated. If the part is not anchored, it will go to an alternate case, which is "Unanchored". You may have observed that the `or` operator acts similarly to the `else` keyword because of the nature of this implicit expression.

Conditional statements are a core component in programming and, as you have seen, have a multitude of applications for even the most basic systems in game development. In the next section, we will cover loops. **Loops** often go hand in hand with conditional statements, since they can feed whole sets of data into a conditional or repeat manipulation as needed to accomplish the behavior you want.

## Declaring and using loops

**Loops** are valuable components when it comes to programming, especially when working with sets of data. It would, of course, be quite unrealistic to expect a programmer to index and assign all one thousand elements of a hypothetical table to variables in order to perform some sort of operation. Or in a Roblox setting, manually change the properties of thousands of parts if you wanted to. To accomplish behaviors like this, loops are key. They function by jumping back to the beginning of their code block if a condition is still met, executing until they reach their terminating case.

### for loops

**for** loops are a type of loop that are primarily used for iterating over datasets. In Luau, those datasets are typically related to tables or numbers. In Luau, there are two types of **for** loops: **numeric for** loops and **generic for** loops. The primary difference between these is what determines how they are executed. For numeric **for** loops, a variable is assigned to a defined start value, end value, and optionally an increment value; if the increment is not included, Luau sets the increment to 1. The numeric **for** loop will then execute the contained code a specified number of times, treating the endpoints of the number range inclusively. Additionally, the assigned variable serves to tell you what the current value of the loop is. Much like the use of `if` and `then` in conditionals, **for** loops use `for` and `do` as their declaration keywords.

The following example prints numbers going from 0 to 10, incrementing by 1 with each loop completion. Note that the increment in this case did not need to be specified but has been shown to aid with your understanding:

```
for i = 0, 10, 1 do
    print(i)
end
```

Let's create a more practical example using a numeric for loop where we find the sum of all integers ranging from 1 to  $n$ . Additionally, we can test that the for loop works correctly by plugging in the same value for  $n$  into the theorem for this operation:

$$\sum_{i=1}^n i = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

Figure 3.5: Theorem for the sum of the first  $n$  natural numbers

In this demonstration, there are some details you should pay attention to. Firstly, since the loop increments by 1, the need for an increment value is not needed; when it is not needed, it should not be included in order to maintain good style. Secondly, you will notice the use of `tostring()`, which functions much like the aforementioned `tonumber()`. This is used because while `print()` will automatically convert other data types into strings, you cannot append other data types, except for numbers to strings. In this example, we will find the sum, print it, and then print whether the sum that was found by the for loop matches the value expected by the theorem:

```
local n = 17
local sum = 0

for i = 1, n do
    sum += i
end

print(sum) -> 153
print("Function working = ".. tostring(sum == (n * (n + 1)) / 2))
-> "Function working = true"
```

Returning to the for loop types, we have generic for loops. While the name seems to imply that they are not useful or special, you will likely utilize them more than numeric for loops – or most other types of loops for that matter. Generic for loops allow you to traverse all indices and values returned by an **iterator function**.

In programming languages, an iterator function is designed to allow programmers to process every element of a data structure while making those returned values isolated from the data structure itself except when the element is passed by reference, rather than by value. We'll cover what this means in the *Recursion* section of this chapter. For now, keep in mind that things like tables and instances are passed in directly where values like numbers or strings are copied. Modifying the former types in the loop will modify them directly. This isolation of copied types, which can be seen when defining a variable in a code block, relates to the concept of **scope**. This means that something that's declared in a block cannot be referenced outside of that block. In the following example, we have a dictionary called `items` that contains three strings. By providing this data structure to the `pairs()` iterator function, you can nicely display every index and value being provided by the iterator:

```
local items = {  
    Animal = "Elephant";  
    Food = "Egg";  
    Plant = "Flower";  
}  
  
for index, value in pairs(items) do  
    print(index, value)  
end
```

As we mentioned previously, the index and value provided by the iterator are not components of the actual data structure that is passed to `pairs()`. This means you are free to manipulate these as desired without the risk of affecting the elements currently being processed. In the following code, the goal is to double any odd numbers to make them even. While we could assign the number that results from using modulo to a new variable, it is alright in this case to simply use the value variable directly. Notice that in order to actually change the element of the table that value corresponds to, you must use the index provided by the iterator with the table itself. In this example, we use the `ipairs()` iterator function. `ipairs()` should be used with tables being used as arrays as it ensures elements are processed in order, where `pairs()` does not. For example, while you could continue to use `pairs()`, elements may be processed in a non-sequential order; for example, the second element (60) before the first element (37):

```
local values = {37, 60, 59, 20, 4, 10, 100, 75, 83}  
  
for index, value in ipairs(values) do
```

```
value %= 2
if value == 1 then --Odd number
    values[index] *= 2
end
end
print(values) -> {74, 60, 118, 20, 4, 10, 100, 150, 166}
```

Our last generic for loop example will be parsing through instances that have been parented to the **Workspace**. Changing the properties of various objects within the Workspace is where you will likely utilize this type of for loop the most. In the following demonstration, we will be looking for and anchoring all BaseParts. By using the `GetDescendants()` method of the Workspace, all instances contained within it are returned recursively, meaning it does not matter where an instance is parented, only that it exists somewhere in the Workspace. For clarification, on Roblox, there is a parent-child hierarchy where the parent of an instance is whatever the instance exists inside of and the instance itself is the child of that parent. It is pertinent to remember that the `GetChildren()` method also exists, which returns a table of only the immediate children of an instance. Notice that the index position has an underscore (`_`) as opposed to a name that could be better referenced; this is merely a stylistic choice for when the index component is not being used, making it more readable for someone who would otherwise be looking for its location in your code. Here, you can see that we check that the object is a BasePart since the `Anchored` property would be nonexistent for any other class of instance:

```
local items = workspace:GetDescendants()
for _, object in pairs(items) do
    if object:IsA("BasePart") then
        object.Anchored = true
    end
end
```

You will now learn about a different type of loop that will always run, so long as a condition is met.

## while loops

**while** loops are loops that run continuously, as long as some specified condition is met. Though they can be used for similar purposes as for loops, it is best to think of them as a repeating conditional statement.

In the following example, the `while` loop increments a value by 1 only if that value is less than 10:

```
local num = 0

while num < 10 do
    num += 1
end

print(num) -> 10
```

In a game development setting, while loops are particularly useful for running round-based games or performing any other event that should occur after a given amount of time. For example, if in a round-based game the round timer hasn't run out, you should do whatever is necessary to keep the game going. Something to remember is that conditions are not restricted to just a Boolean value. As long as the condition is not **falsy** (false or nil in Luau), the loop will execute, and if the condition itself is a function, that function will execute and the loop will also run if the function returns a value. This is also true of the condition of conditional statements. You should remember to use good style when doing this, which we will discuss more in the *Demonstrating programming style and efficiency* section.

Another variation of the `while` loop is the `while true` loop. This type of loop will always execute as the condition is always true. This unending variation of a `while` loop is useful for something that does not need to terminate; however, it can cause a script to crash as the loop stacks on top of itself infinitely. To avoid this, we can use the `task.wait()` function; this function serves as a pause within your script, which is convenient for delaying a behavior or allowing more time for something to occur. Adding a `task.wait()` to this loop will prevent the loop from stacking and instead run at the rate you want. In the following code, a `while` loop has been created that increments a variable every second, keeping track of the elapsed time since the loop began. Note that the number argument to `task.wait()` is in seconds:

```
local timeElapsed = 0

while true do
    wait(1)
    timeElapsed += 1
    print(timeElapsed)
end
```

In the next section, you will learn about a similar type of loop that can be used in slightly different applications.

## repeat loops

**repeat** loops execute their contents until a condition is met. While this may seem much like a **while** loop, the difference is that **while** loops run only if a condition is met, checking the condition before running. Unlike other loop types, repeat loops always run at least once, checking the terminating condition only after execution, much like a **do-while** loop in other languages. The keywords for repeat loops are **repeat** and **until**, where the code to be executed follows the **repeat** keyword, closed by **until**, and ends with the condition to leave the loop. The following loop shows a number variable being decremented until its value is equal to 0:

```
local num = 12
repeat
  num -= 1
until num == 0
print(num) -> 0
```

With loops now at your disposal, you can process large sets of data and make systems that require consistent, repetitive behavior. Next, you will learn about a new way of feeding data to loops, as well as condensing them if they are frequently used.

## Learning about functions and events

In programming, a function is a repeatably callable code block, typically designed to accomplish a single task. Functions are important for abbreviating common jobs being done and help reduce the amount of redundancy within your programs. In this section, you will learn different ways to format functions, as well as when you should be using them.

### Functions in programming

We primarily use functions to define code that can be easily referenced and executed repeatedly. For the sake of terminology, many programming languages distinguish functions from **procedures** or **subroutines**; the difference here is that a function executes code to compute some data that is returned, whereas a procedure simply accomplishes a task without returning a value to where it was called. The following function has been designed to create a new table and fill it with fruits, vegetables, or a non-produce item based on a randomly generated value. See that the function is locally defined, much like a variable, followed by the **function** keyword and the name of the function, and ends with a set of parentheses (**()**); this part of a function is called the **header**. To select a random item, we make a new random object using the **Random.new()** constructor; this is created in a manner quite similar to **Java**.

By using the `NextInteger()` method of the random object, a random integer can be generated between the min and max value that's provided to it inclusively. This function could be associated with one of the examples seen in the *Conditional statements* section. Notice how we declare a new variable `item` but do not assign it a value. This is proper syntax and the variable will hold a value of `nil` by default. If you feel comfortable, try making a produce counter using the conditional statement from the previous section, a loop, and this function:

```
local random = Random.new()
local function fillStoreSupply()
    local storeSupply = {}

    for i = 1, 10 do
        local ranVal = random.NextInteger(1,3)
        local item

        if ranVal == 1 then
            item = "Fruit"
        elseif ranVal == 2 then
            item = "Vegetable"
        else
            item = "Shoe"
        end
        table.insert(storeSupply, item)
    end

    return storeSupply
end

local supplyTable = fillStoreSupply()
```

One of the main aspects of using functions is providing information to them when you call them for a task. To do this, values need to be added to the call statement of the function and defined in the line where the function is declared. When a value is being provided to a call, it is referred to as an **argument**. However, when referring to this data inside a function, it is referred to as a **parameter**. The following function creates a factorial from the provided number, *n*. A factorial is the result of multiplying all whole numbers less than a number, by that number. See how a number is provided as the argument in the function call.

When the function runs, that value is automatically assigned to  $n$ , which can then be manipulated as needed:

```
local function factorial(n)
    assert(n == math.floor(n), "n must be a whole number.")
    local factorial = 1 --Empty product should be 1

    while n > 1 do
        factorial *= n
        n -= 1
    end

    return factorial
end

print(factorial(12)) -> 479001600
```

You may have noticed the use of the `assert()` function. Much like the use of `throw()` in Java, you can use this to throw an error and terminate a process if some condition is not met. The second argument is a string that is sent to the output and will look like any other naturally occurring error message.

In the case that you do not know how many arguments are going to be passed to a function, you can create a **variadic function**. Variadic functions are like regular functions, though they possess the ability to take any number of arguments in a tuple state. The following variadic function returns the sum of all numbers provided to it. Notice the use of the three dots (`...`); these dots represent whatever arguments are passed to the function and are most often put directly into a table for processing. In the following code block, you can see a random amount of number arguments being passed to the `sum` function. The parameters are added and returned as a single value:

```
local function sum(...)
    local args = {...}
    local sum = 0

    for _, number in pairs(args) do
        sum += number
    end
```



```
        return sum
    end

    local num = sum(7, 9, 12, 3, 2, 6, 13)
    print(num) -> 52
```

As you may have found out on your own, all the loops we have covered have the ability to **yield**; that is to say that when they run, they pause the current **thread**, meaning that the loop must finish before any code following it can be executed. In programming, we can make use of what is called **multithreading** to handle this. Luau does not yet support true multithreading though it is currently being developed and should release in 2022. For the time being, the `task.spawn()` function is used to create what we can consider a new thread within a script. In the following example, a `while` loop is running every 1 second in order to record the elapsed time. Note that this loop has a behavior that's equivalent to the second example that was provided in the *while loops* section; though the condition is different, it is not `false`. Normally, code after this `while` loop can never be reached as there is no condition that will cause the loop to end. However, with it wrapped within a `task.spawn()` function, a new thread is created just for the loop or any other contained code, and the rest of the script's content in the main thread will not be impeded from running:

```
local timeElapsed = 0

task.spawn(function()
    while task.wait(1) do
        timeElapsed += 1
        print(timeElapsed)
    end
end)

print("Code after loop can still execute!")
```

The next section will cover recursion, a useful technique for solving some types of problems.

## Recursion

One of the invaluable features of functions is their ability to call themselves. When properly structured, this can create what you might think of as a loop in a process called **recursion**. The difference between something like a `while` loop and a recursive process is that a loop jumps back to its beginning, whereas recursion actually **stacks** upon itself.

In programming, a stack can be a data structure or, as in the case of recursion, simply the state of something in your program. Like a stack of plates or pancakes, the one that was most recently added will be the first one to be removed.

To demonstrate this stacking, let's return to the factorial function we created earlier in the *Functions in programming* section. Though a while loop was able to accomplish the goal, you could also achieve the same result by using recursion. In the following function, notice that the call and header of the function remain unchanged; the recursive elements exist in the return statements. For the if statement, the first case simply returns 1 if  $n$  is less than 1, because we cannot create a factorial from any values less than this; in the case of  $n$  being 0, its factorial would also be 1 by convention. The next case is the most important: if  $n$  is greater than 1, then it is multiplied by the value that's returned by the factorial function, where  $n$  is one less than the current value of  $n$ . As you may begin to see, this causes the function to stack until  $n$  has been decreased down to 1. This is the **base case**, where no function call is made, and we work back down the stack:

```
local function factorial(n)
    assert(n == math.floor(n), "n must be a whole number.")

    if n <= 1 then
        return 1
    else
        return n * factorial(n - 1)
    end
end

print(factorial(6)) -> 720
```

To give a better visualization of how this process is being executed, let's look at a mapped-out example of the previous call to the factorial function, using 6 for  $n$ . Observe that each call,  $n$ , is set to be multiplied by the returned value of the function and continues to be stacked until a case without a function call is reached. Then, each function stops and is taken off the stack, returning the value to the return statement that called it. Once this process finishes, our original function returns the final value to wherever it was called from:

```
factorial(6)
6 * factorial(5)
6 * (5 * factorial(4))
6 * (5 * (4 * factorial(3)))
```

```

6 * (5 * (4 * (3 * (factorial(2)))))
6 * (5 * (4 * (3 * (2 * factorial(1)))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
720

```

Now that you have a firmer grasp of the concept of recursion, let's look at another practical example. When working with tables, setting a variable to an already existing table will not follow normal behavior and simply copy that value to the new variable; instead, tables use **references**. References work to save resources, essentially causing new variables to act only as pointers to a previously declared table; the pointer can actually be seen by printing the table. With this behavior, assigning a table to a variable and changing anything within that table would change it everywhere it is referenced. You can test this with the following code. Here, you can see that when a variable's value is set to a table that has already been created, the variables contain the same reference:

```

local function checkEquality(table1, table2)
    print("Variable 1: ".. tostring(table1))
    print("Variable 2: ".. tostring(table2))
    print("First and second variable same table = ".. tostring(table1 ==
        table2))
end

local group = {"Ashitosh", "Grey", "Manish", "Meenakshi"}
local groupClone = group
checkEquality(group, groupClone)

```

If a table were cloned every time it was assigned to a variable, that would make indexing libraries or any other large table structure extraordinarily expensive. There are scenarios, however, where you may need to clone a table or dictionary. While for loops may be viable in some cases, the presence of nested tables, as seen at the end of the *Dictionaries* section, could potentially require that you use any number of for loops to accomplish your task. To get around this, we can once again use recursion. The following example creates a copy of our items table by creating a new table and adding each element to it by index and value. In the case that the value to be cloned is a table, the function recurses with the nested table as the argument.

Once it's done this, the completely new table is returned to where it was called from, and the `checkEquality()` function from the previous example is used to verify that the tables are unique:

```
local items = {
    Egg = {fragile = true;};
    Water = {wet = true;};
}

local function recursiveCopy(targetTable)
    local tableCopy = {}

    for index, value in pairs(targetTable) do
        if type(value) == "table" then
            value = recursiveCopy(value)
        end
        tableCopy[index] = value
    end

    return tableCopy
end

local itemsClone = recursiveCopy(items)
checkEquality(items, itemsClone)
```

Like loops, calls to recursive functions can yield, but typically, they finish in a short enough amount of time to where nothing in your thread is affected. If, for some reason, your recursive function runs long enough to cause a noticeable pause for the rest of your script, you should review your function rather than try to implement a new thread. This is because any code following the new thread will execute before the value is returned, causing potential issues.

## Events and methods of instances

Just like instances have different properties depending on their class, they also have unique methods and events that vary with their class. Just like base classes grant the same properties to multiple instances, some methods and events are inherited by all instances, or likewise by base classes.

For the following examples, we will continue to use a part as our instance of choice. The following method returns the mass of that part, taking into account the density of the part based on its `PhysicalProperties` and its size.

Using this, we can calculate the amount of force needed to locally simulate zero gravity for this object. We will be using a `VectorForce` instance to accomplish this; we will cover the `Constraint` base class and more of its types in the *Working with physics* section of *Chapter 4, Roblox Programming Scenarios*. To make this work best, place your part slightly off the ground and jump on it or apply some other force to see it start moving. Make sure the part is unanchored; otherwise, it will not be able to move:

```
local part = workspace.FloatingPart
local attachment = Instance.new("Attachment")
attachment.Parent = part
local mass = part:GetMass()
local vectorForce = Instance.new("VectorForce")
vectorForce.Force = Vector3.new(0, mass * workspace.Gravity, 0)
vectorForce.Attachment0 = attachment
vectorForce.Parent = part
```

While methods have already been well defined, **events** are best described as detectable signals that are sent out when something happens within your experience. These signals can be easily connected to functions, which allows you to associate any event that is triggered with a predefined behavior in an easily manageable cause-and-effect relationship. When using the appropriate terminology, when a signal is triggered, we say it has been **fired**.

For this demonstration, we will be examining the `Touched` event of `BaseParts`. This event is fired whenever another `BasePart` touches `TouchPart`, the part looking for the event's signal. For connecting signals to functions, we use the `Connect()` method. Much like how a `task.spawn()` function is formatted, you can include the `function` keyword and a set of parentheses `()`. However, many events pass information in the form of arguments that you will want to include in your function. You can define these parameters like you would when creating your own function to be called. Here, the part that has been touched is an argument that's passed by the event. It must be defined in the function header if you wish to use it:

```
local part = workspace.TouchPart
part.Touched:Connect(function(hit)
    print(hit)
end)
```

In the preceding example, the function has been formatted so that it is embedded within the event. For many, this is the preferred method of doing things, but if you want a function that is being used elsewhere to also be associated with an event, you can provide the function name to the `Connect()` method. In the following example, we are locally defining a function and, separately, declaring that the function should be called when the touched event is fired. Notice that the function's first parameter is still present without us needing to pass it from where the event is connected:

```
local part = workspace.TouchPart
local function printHitName(hit)
    print(hit)
end

part.Touched:Connect(printHitName)
```

You have learned a lot of information in this chapter that will be relevant to you in the future, just as much as it is relevant to you now. While you now know enough from these sections to start experimenting with making some intermediate programs, like a moon gravity script or the beginnings of obstacle course hazards, you must learn how to do so with good style and make sure that your code is as efficient as possible.

## Demonstrating programming style and efficiency

Writing code with good style not only improves the quality of your work, but it also prepares you to pursue programming in more professional environments or when working with other people. The rules you are about to cover are categorized by whether they exist universally in programming or if they pertain primarily to Roblox.

### General style rules

**Readability** is an important aspect of maintaining good style. Not only do others who may read your code need to understand what is happening, but being able to easily follow your own code will greatly increase your workflow. Having a clean coding style will also enable you to be more conscious of other style factors you should be implementing. The two ways you can make your code the most readable are to use proper indentation and observe appropriate line length. For line length, most college programming courses will likely suggest that you limit your lines of code to eighty to one hundred characters.

In Roblox, there are line numbers but not an indicator for which column you are on for a line. Generally, your line length should not exceed your viewport size, which means that you should not require the use of a horizontal scrollbar to see the entirety of your line. Out of all the readability rules, you should arguably observe indentation style the most carefully. Roblox Studio should automatically indent your code when you hit the *Enter* key, but make sure what you write follows the code examples in this chapter until you feel confident with your ability to follow this rule.

Going back to working with multiple programmers, comments are an invaluable part of letting others, as well as your future self, know what your code is doing. While readability is also needed to make others aware of what a script's purpose is, comments can be used to more explicitly tell fellow programmers where a code block is being used, what it requires, or what behavior to expect from it.

Looking back at implicit conditionals, there are situations where it may make more sense to use logic within a variable declaration than to use an explicit conditional statement. This decision is ultimately up to you, but only use it when it's practical. If you need to create more than two or three cases, you should likely just use a conditional. Additionally, remember to consider previously mentioned style points, such as line length.

As mentioned briefly previously, you should ensure that you are not grouping random data together in tables or dictionaries. For organizational purposes, you should be using tables for a significant purpose, and the elements of data within them should be at least loosely associated with each other. If necessary, there is no harm in creating additional tables to accommodate different sets of data being used in your code.

A **naming convention** refers to how you format the names of your variables. There are many conventions; the first two you'll likely encounter being Pascal case and camel case. Camel case is where the first letter of the variable is lowercase with the first letter of any other words in the variable being capitalized; for example, `camelCase`. Pascal case is where the first letter of all words in the variable name are capitalized; such as, `PascalCase`. While this is up to your preference, you should stylistically use Pascal case when programming on Roblox.

## Roblox-specific rules

Optimizing code is very important when making sure every player in your potential audience does not experience issues when playing. While we will cover a variety of ways to make different systems as efficient as possible, you should learn to implement events as opposed to loops whenever possible.

For example, if you know that a value that's being displayed to the player is subject to change, you should use an event to detect this rather than constantly updating the displayed information with a loop. This is so that resources are only being used when the event is fired, as opposed to constant usage.

Some Roblox methods and functions have certain disadvantages that cause them to be labeled as **deprecated**, meaning that while they still work, they are not recommended for use in new work, nor do they show up in autofill. None of the examples in this book contain deprecated content at the time of writing and if you are unsure when writing your own programs, you are encouraged to view the documentation for what you are about to use via the developer website.

By making use of the rules we've covered, you can increase your abilities as a programmer exponentially. Not only will those you code with be more appreciative of your work but in the following chapters, writing programs more effectively will allow you to better grasp new material and implement that material in new coding creations.

## Summary

In this chapter, you learned about programming constructs that exist in a wide variety of languages, such as variables, data types, loops, and some data structures, as well as those that are exclusive to Luau. With this knowledge, you can begin making your own mental connections by experimenting with the examples from this chapter and making your own programs.

In the next chapter, you will learn about programming that pertains almost exclusively to Roblox development. Following that, you will utilize the new information you have learned about to start making your first game-oriented systems, which will lead to you making full experiences in the following chapters.

## Worksheet

- Q1. What is used in programming to hold a piece of data?
- Q2. What simple function can be used to send information to the **Output** window?
- Q3. What data type should you use if you want to represent a value of true or false?
- Q4. What data type is used if you want to represent a word or characters?
- Q5. What is used in programming when we want to check if a condition is met?
- Q6. What is used in programming when some action must be done repeatedly so long as a condition is met?



- Q7. What loop would be best to use if we want something to happen an exact number of times?
- Q8. What type of loop is best if we want something to always happen as long as some condition is met?
- Q9. What type of loop is best if we want to do something immediately once and then loop only while some condition is met?
- Q10. What do we call a repeatedly callable block of code that is typically designed to accomplish some single task or part of a task?
- Q11. What is the part of a function called where the function's name and parameters are present?
- Q12. What type of function can take any number of arguments without them being explicitly defined in the function's header?
- Q13. What is a function said to be if it calls itself?
- Q14. What is a part of an instance that we say is "fired" when something happens?
- Q15. What is a part of an instance that is callable and causes some behavior?
- Q16. Why should you use good programming style no matter what?

# 4

## Roblox Programming Scenarios

While you have learned many aspects of programming in a general sense, some scenarios apply almost exclusively to game development using Roblox. This chapter will explore Roblox-specific programming, Roblox services, how to manage players, physics manipulation, and other experience features.

In this chapter, we're going to cover the following main topics:

- Understanding the client-server model
- Using Roblox services
- Working with physics
- Adding peripheral experience aspects

Let's get started!

### Technical requirements

Like the previous chapter, you will be working entirely in Studio. You will need an internet connection to use Roblox Studio, as well as to access the developer website to further explore some of the topics that will be covered. You can find all the code used in this chapter in the book's GitHub repository at <https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy-2nd-Edition/tree/main/Chapter04>.

Additionally, you can find the Roblox developer website at <https://developer.roblox.com/en-us/>.

## Understanding the client-server model

The **client-server model** is a distributed communication structure that can be found throughout many fields of computing. In game development, this type of communication is most often used so that the server acts as a provider and verifier for clients who make requests to it. In other words, the server is a computer that is responsible for holding and dispatching information and the client is someone or something (a computer application or a player of your experience in the Roblox sense) that asks the server for information or requests the server to perform some action. In this section, we will learn how to work with both sides of this model as a programmer on Roblox.

## Different script types

Since the server and client are separate models with different defined purposes, programming in Roblox requires the use of unique instances from the `LuaSourceContainer` base class, depending on whether a script is working with the client (local) or with the server. For clarification, the `LuaSourceContainer` base class includes all instances that are made to hold or execute Luau code on Roblox. A third type of script exists that can be accessed locally or by the server. These script types and their icons can be seen in the following screenshot:

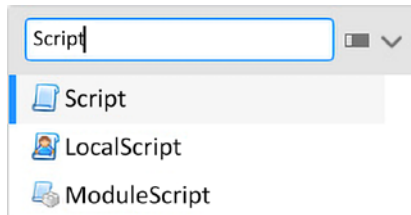


Figure 4.1: Different script types can be used based on which data model you are working with

Let's look at each script type and its applications in the following sections.

## Scripts

**Scripts** are used to execute code when working with the server and were likely the first script type you were introduced to. To run, they must be a descendant of the **Workspace** or **ServerScriptService**, as shown in the **Explorer** window. Scripts contain `print("Hello World!")` by default. Because scripts operate on the server, they are most often used for overall experience management; examples may include setting up a player's data or changing aspects of the experience's environment.

Scripts will continue to execute until an error is produced, the end of the thread is reached, or they are parented to anywhere they are not permitted to run. Additionally, if a script or the parent of a script is destroyed, the script will be terminated as a result. Since these server-side scripts cannot be influenced by individual clients, some Roblox services allow them to use more of what they have to offer; this will be covered more in the *Using Roblox services* section.

## Local scripts

**Local scripts** are like server-side scripts but are used when working with the client. Local scripts are the only way to obtain local information about a player, such as the CFrame of a player's camera, mouse properties, physical input, and more. They can only be executed if they are parented to a player's Backpack, character, PlayerGui, PlayerScripts, or the ReplicatedFirst service. All of these will be discussed in more detail in the *Using Roblox services* section of this chapter.

Client scripts have vulnerabilities, in the sense that they can be created and executed by clients with **script injectors**, which is a tool in which they can insert local scripts that execute whatever code they choose. As a result, local scripts do not share all the same permissions to use Roblox services as server scripts do. Since local scripts are dependent on a client, they can identify which client is executing them. For example, if we wanted to return information about a client's mouse, that can be done from a local script; you can assume that the `player` variable is already defined as a player instance:

```
local mouse = player:GetMouse()
print(mouse) --nil in server script
```

## Modules

**Modules** are a type of script that must be accessed by using the `require()` function. Code that is contained within modules will be unable to run until this function is called and, in turn, the returned result is cached. This means that the code in the module itself will not be run multiple times if the `require()` function is repeatedly used, and the module table will always have the same reference.

The primary use of modules is to hold functions or other code that can be easily referenced globally rather than isolated to a single script. You may have seen the global identifier (`_G` or `shared`) for defining variables or functions, but it is generally considered poor style to use this. Modules can be accessed by either the server or the client, making them useful for storing functions you want to be accessible to both; just be sure to keep them organized.

In the following example, we have a module holding a simple function and a separate script calling that function, as denoted by the comments in the code. Notice that the module returns the table where information is stored; this must occur in modules as the previously mentioned `require()` function essentially calls the module to return its contents:

```
--module
local module = {}
module.initialize = function()
    print("Initialized")
end
return module
--script
local mod = require(module)
mod.initialize()
```

Note that in this example, we've decided to make the function *initialize* a member of the module via a period (.) before the function name. You can alternatively define a function to be a member with a colon (:) before the function name but in this case, the module table (known as `self`) will always be passed as the first argument to that function. When defined like this, you also put a colon rather than a period when calling the function. Both are valid forms and should be used depending on your needs. For this book, we will be using a period to define member functions.

While modules can be created to act as classes, holding functions and other code to be executed, they can also be used to hold tables of data, typically using a string as a key. To demonstrate this, we will use NPC properties as an example. Using a loop, you could set up every NPC in a folder by using their name as an index and then set the `Humanoid` properties to the corresponding values that are in the module table. Notice in the following example that the script can index values in the module as if it were a table in the script directly:

```
--module
local module = {}
module.Heavy = {
    MaxHealth = 500;
    Health = 500;
    WalkSpeed = 11;
}
return module
--script
```

```
local mod = require(module)
print(mod.Heavy.MaxHealth)
```

Now that you have learned about the different script types, you must be aware of the tools available to you when working inside them.

## The Script Menu tab

The **Script Menu** tab of Studio is a menu only visible when you're inside a script instance; you can see this menu in the following screenshot:



Figure 4.2: The Script Menu tab provides many tools while you're working in scripts

The **Navigate** section of the **Script Menu** tab allows you to move forward or backward through the scripts you currently have open in Studio. You can also simply double-click scripts that are open to return to them or click the associated tab.

The **Edit** submenu provides several different functionalities, the first being the **Find** tool. While there are many options you can use with this tool, the most commonly used are the standard **Find** action (*Ctrl + F*), **Find All** (*Ctrl + Shift + F*), and **Go to Line** (*Ctrl + G*) tools. The **Find** action is straightforward and simply looks for strings in the current script that match your input. **Find All** searches all the current script instances within your experience for strings that match your input; the results will be returned to you in a new in-Studio window, much like the output window. Finally, **Go to Line** will bring up a new modal window with a box that asks for you to input a line number. Once inputted, your line selection will move to the specified number and the box will close.

The **Replace** action will bring up a menu in your script, similar to that of the **Find** tool, and will prompt you to provide two strings. The first string is what you want to find in your script, while the second is what you want that string to be replaced by. Unfortunately, this feature is limited to working with one script at a time, and you cannot replace all desired strings within all the scripts of your experience at once.

Roblox additionally provides a variety of tools to aid in debugging. For this, you need to be introduced to the concept of breakpoints. **Breakpoints** will pause your test session once the line of code they are associated with is reached.

To add a breakpoint, you can simply click the space to the right of a line number when inside of a script, as shown in the following screenshot:

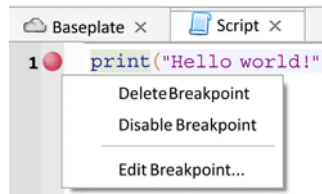


Figure 4.3: You can insert breakpoints to stop your code when a line is reached

Once a breakpoint has been reached, you can continue through the code line by line by using the **Step Into**, **Step Over**, and **Step Out** options under the **Debugger** submenu, as shown in Figure 4.4. **Step Into** will continue through your code line by line, entering any functions or blocks of code that exist elsewhere that are referenced. The **Step Over** action will skip over any code blocks if the next line would otherwise bring you into one. Finally, the **Step Out** action will take you out of a code block immediately if you have reached one, bringing you to the next line following the container:

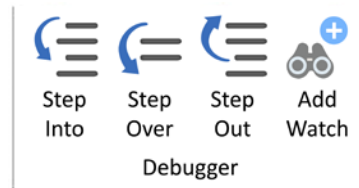


Figure 4.4: The Debugger tool has different actions you can use to progress through code

Under the **View** tab of Studio, you can also find the **Watch** and **Call Stack** windows. Both serve to provide additional information about what your code is executing when debugging. By toggling the **Call Stack** view, you can see which processes are currently on the stack; that is, you can see if you are currently in a function, view the order in which functions are called, and see which lines those calls are made from. You may find this feature particularly useful if you are working with a recursive function, as you can see the order of calls. To aid you as you follow your code, you can also toggle the **Watch** window to see the exact values of the variables and types of expressions within your script.

Look at the advantages of using both tools to follow a recursive function, similar to the one shown in the following screenshot:

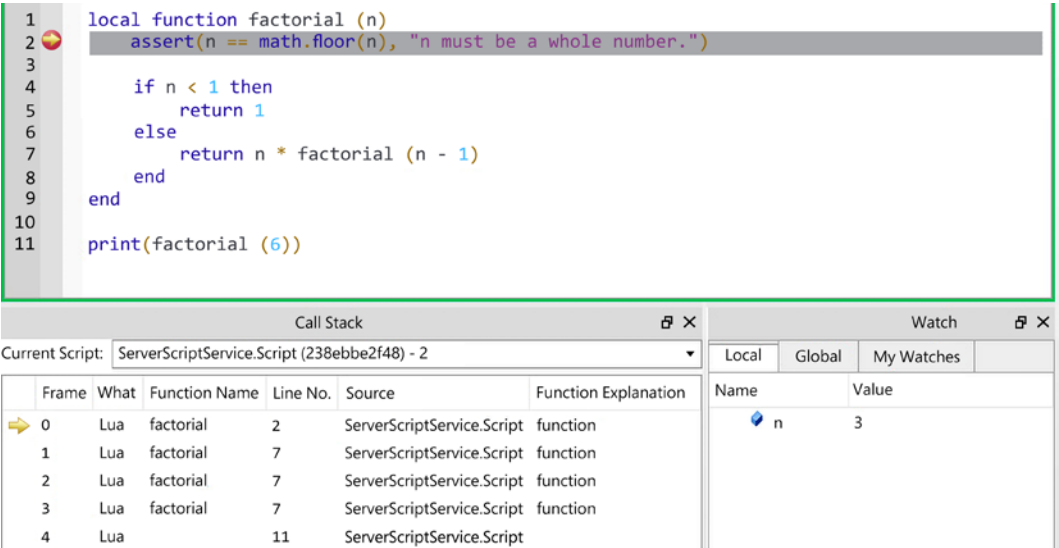


Figure 4.5: The Watch and Call Stack windows are helpful for following complex code

Lastly, to assist you with style, you can use the **Format Selection** button to properly indent your code. This feature was recently added and allows you to indent only the code you currently have selected or the entirety of your script. Indentation is automatically ascertained from the location of the keywords for code containers.

## FilteringEnabled

In 2018, Roblox made the **FilteringEnabled** setting always set to enabled for all experiences to prevent exploitation from occurring within them. **FilteringEnabled** essentially serves to force a client-server communication structure as opposed to the previous, less verifiable communication and replication method, where clients could directly tell the server what behavior should occur. As a result of this major system change, many existing experiences were no longer playable because they were not programmed to accommodate this communication style. Today, all projects on Roblox are scripted with **FilteringEnabled** in mind, making use of specific instances, called remotes, to support communication between the client and server.



## RemoteEvents

A `RemoteEvent` is a way for a client to communicate with the server and vice versa and are used when no response from the requestee is needed. Often, clients send signals to the server so that the server can fulfill a task on their behalf. To do this, the client must **fire** a `RemoteEvent` using the built-in method. It should be noted that this method can only be used when you're working in a client script, so your `RemoteEvent` should be parented to somewhere where it can be easily accessed by the client and server, such as the `ReplicatedStorage` service, which we will cover in more detail later. In the following example, you can see that a client is calling the `FireServer()` method of a pre-defined `RemoteEvent`:

```
RemoteEvent:FireServer()
```

Once the `FireServer()` method causes the remote to be fired by the client, the signal can only be detected on the server. By using a `RemoteEvent` and the standard `Connect()` method, you can create whatever desired behavior you want to occur on the server. Calling `FireServer()` will never cause the client to yield, even if a call to `task.wait()` is present in the server function because the client and server scripts are two separate threads on separate models. Additionally, the client that fired the remote is passed as an argument to the event function when the event is fired; additional arguments can be provided in tuple fashion:

```
--client
RemoteEvent:FireServer("Hello world!", 3);

--server
RemoteEvent.OnServerEvent:Connect(function(player, message, num)
    print(message) -> Hello world!
    Print(num) -> 3
end)
```

This one-way method of communication via a `RemoteEvent` can be reversed so that the server creates a signal that can only be detected by the client; this signal can be received by the client and similarly execute any desired function. Two methods can be used to accomplish this. By calling the `FireClient()` method of a `RemoteEvent` on the server, the specified player will have an event sent to them. Alternatively, you can fire all the clients within the experience, as opposed to a specific one, by using the `FireAllClients()` method. This is often useful for global events or notifications rather than looping through individual clients.

Arguments can still be provided to the client as a tuple:

```
RemoteEvent:FireClient(player) --Sends signal to single client
RemoteEvent:FireAllClients() --Sends signal to all clients
```

Much like using `OnServerEvent` when processing a client request, clients can detect these signals in local scripts by using `OnClientEvent` and, similarly, associate them with a function. In the following example, a client is receiving a signal from a `RemoteEvent` that has been fired by the server. Here, you can see that this remote is sending a signal to all the clients to accomplish a task that can only be done locally, sending a default notification using `StarterGui`, a part of Roblox's services. This will be covered in the *Using Roblox services* section:

```
--server
wait(5)
RemoteEvent:FireAllClients("Game over!", "Blue team has won!")
--client
local starterGui = game:GetService("StarterGui")
RemoteEvent.OnClientEvent:Connect(function(title, text)
    starterGui:SetCore("SendNotification", {
        Title = title;
        Text = text;
    })
end)
```

A very common use case for a `RemoteEvent` is for a client to fire one to damage another player. These damage remotes will most likely be required for all the weapons in your experience if your hit detection is being done on the client. Hit detection on the client is necessary in most cases, as **latency** (the delay in communication between a client and the server) makes its use on the server unreliable. If a client is injecting their own code, they can provide different arguments to a `RemoteEvent`. Consequently, you should make sure that you are getting information from the server whenever possible. Do not let clients tell the server how much damage should be dealt or the weapon they are currently using. Checking that the values provided make sense (sometimes known as **sanity checks**) is the best way to protect your experience in this situation.

## RemoteFunctions

A `RemoteFunction` is similar to a `RemoteEvent` but has the ability to return information across the client-server boundary. The primary use of this type of instance is to retrieve information that would normally not be accessible to the server from the client or vice versa.

In the following code, a client is making a request using the `InvokeServer()` method of a `RemoteFunction`:

```
RemoteFunction:InvokeServer()
```

Formatting the function that receives a signal from a `RemoteFunction` invocation is slightly different than when using a `RemoteEvent`. A `RemoteFunction` will not use the `Connect()` method to link its signals to functions. Instead, a function is directly assigned to them, called a **callback**. In the following code, the server is receiving the signal from the client's previous request by using the `OnServerInvoke` event. Notice that like `RemoteEvent`, the client that made the request to the instance becomes a parameter of the assigned function:

```
RemoteFunction.OnServerInvoke = function(player)
    return --optional but use a RemoteEvent for subroutines
end
```

The direction of communication can once again be switched, similar to a `RemoteEvent`, to create requests to the client from the server. For this behavior, a `RemoteFunction` will use the `InvokeClient()` method:

```
RemoteFunction:InvokeClient(player)
```

The following code shows how the request that was sent can be received by the client. The `OnClientInvoke` callback of a `RemoteFunction` is used to assign a function to a request. Here, we can see that the server is requesting the `CFrame` of the client's camera, which normally cannot be accessed by the server. Notice that the function that's been assigned to the event can still be defined in other places. It is important to note that while you can assign many separate functions to a `RemoteEvent`, you can only assign one callback to a `RemoteFunction` instance; attempting to assign another will simply overwrite the previous assignment:

```
--server
local clientCamCFrame = RemoteFunction:InvokeClient(player)

--client
local cam = workspace.CurrentCamera
local function getCamCFrame()
    return cam.CFrame
end

RemoteFunction.OnClientInvoke = getCamCFrame
```

Much like checking what arguments a client sends to a `RemoteEvent`, requesting information from a client using a `RemoteFunction` can also be compromised if a player is using a script injector or any other means to exploit. To resolve this, ensure that you employ checks and only request information from the client when needed.

## BindableEvents and BindableFunctions

A `BindableEvent` and `BindableFunction` are much like their `RemoteEvent` and `RemoteFunction` counterparts, except that when they are fired, their signals can only be detected by the data mode that fired it, which can be either the server or the client. This has useful applications if you want to make your own signals for functions to be connected to, or if you want scripts to directly communicate with each other.

To demonstrate how we can apply such instances, we will just look at a `BindableFunction`. In the following code, we are effectively calling a function that only exists in a server script, separate from the one we are invoking the remote from. Notice that with the use of a `BindableFunction`, we can retrieve a value that's been returned by a function without directly indexing what function we wish to be called. The keywords that are used for invocations and assigning a callback to a `BindableFunction` are nearly identical to when using a remote, those being `Invoke()` and `OnInvoke` for `BindableFunction`, and `Fire()` and `Event` when using a `BindableEvent`. Take a look at the following output, which shows what happens when these two server scripts communicate with each other to accomplish a task:

```
--script 1
local function sum(...)
    local sum = 0
    local nums = {...}

    for _, num in pairs(nums) do
        sum += num
    end

    return sum
end

BindableFunction.OnInvoke = sum

--script 2
local sum = BindableFunction:Invoke(2, 4, 11)
print(sum) -> 17
```

With that, you have learned how different types of scripts work between the client and server and how to facilitate communication in a `FilteringEnabled` environment. Moving forward, you will learn about the additional functionality you can introduce to your programs by making use of Roblox services.

## Using Roblox services

With your programming experience so far, you have created scripts that are limited only to the functionality of built-in language features or the methods and events of instances. In Roblox, the use of **services** is necessary to add more complex behaviors and make use of additional events and methods that are needed within your experience. You can think of these services as libraries that are used to work with different aspects of your experience rather than specific data types.

### Players service

Virtually every experience makes use of the **Players service**. The Players service is visible within the **Explorer** window and is where all player instances are kept. The `PlayerAdded` event is a very commonly needed event and is used to identify when a player joins an experience server. The following code prints the name of any player who joins the experience with a small message attached. Notice that the player who joins the experience is provided as a parameter to the function associated with the event:

```
local playersService = game:GetService("Players")
playersService.PlayerAdded:Connect(function(player)
    print(player.Name.. " joined the game.")
end)
```

A counterpart event also exists for checking when a player leaves the experience: the `PlayerRemoving` event. Note that this function processes before the player instance has been removed from the service, meaning you can manage or retrieve information from the player instance, which is passed as a parameter to the function:

```
playersService.PlayerRemoving:Connect(function(player)
    print(player.Name.. " left the game.")
end)
```

While you may elect not to show a player's stats to everyone in a server, doing so may inspire competition and keep people playing for longer. To do this, Roblox has a built-in system called `leaderstats`, which many experiences use to create leaderboards as it does not require additional UI features for it to be implemented.

In the following screenshot, a player's **Gold** stat is displayed in the player list using the leaderstats system. If there were more players in the experience, players' names would be displayed in the order of their stat level, from highest to lowest:

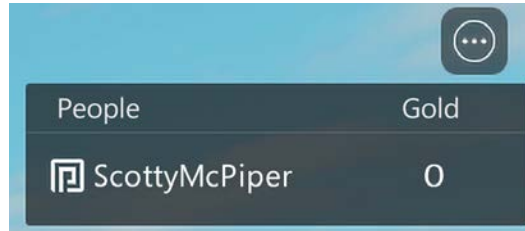


Figure 4.6: You can display information in the player list by using the leaderstats system

Let's look at how we can create our own leaderboard for when someone new joins the experience. Each player must have a leaderstats folder for the system to work, so it is best to use a PlayerAdded event to add the requisite elements to a new player instance. While any instance can be used to hold the name leaderstats, it makes the most sense conceptually to use something such as a folder. Once created and named, the folder should be parented directly to the player instance that's provided to us by the event. Next, create value instances based on which data type you want to be displayed; in the following example, we're creating an IntValue that represents how much Gold the player has. While any number of values can be added to the leaderstats folder, only the first four will be displayed. Moreover, player names will only be sorted in the list based on the first stat that is created:

```
playersService.PlayerAdded:Connect(function(player)
    local folder = Instance.new("Folder")
    folder.Name = "leaderstats"
    folder.Parent = player

    local gold = Instance.new("IntValue")
    gold.Name = "Gold"
    gold.Parent = folder
end)
```

Another common use of the Players service is when working on the client. In order to identify which player is executing a local script, you should index the LocalPlayer property of the service. If you were to index that property inside of a server script, it would be nil.

Once the local player has been indexed, many functions and additional information become available:

```
local player = playersService.LocalPlayer --nil in reg. script
print(player.UserId) --ID unique to each Roblox user
```

For a final example, we will be using the `GetPlayers()` method of the `Players` service in order to iterate over all the clients currently in the experience. The following code uses a loop to set the `WalkSpeed` property of all the players in the experience to 30. It should be noted that a loop structured such as this should not be made to yield unless certain checks are in place. This is because a player can leave in the time it takes for the code to execute, thereby causing problems as that client would still exist in the table of players that's returned by the `GetPlayers()` method but not within the experience itself:

```
local players = playersService:GetPlayers()
for _, player in pairs(players) do
    local char = player.Character
    if char then
        print(char.Name.. "'s speed is now 30.")
        char.Humanoid.WalkSpeed = 30
    end
end
```

Next, we'll look at services used for storing assets of your experience.

## ReplicatedStorage and ServerStorage

The `ReplicatedStorage` and `ServerStorage` services are convenient places to store and manage assets such as models, tools, and more within your experience. Each location has different behaviors and optimizations when content is added to them, but neither of these services has unique methods or properties.

The assets in `ReplicatedStorage` can be accessed by the client but are not visually rendered, allowing for some performance increases when unnecessary assets are moved there in a live experience setting. As we mentioned previously, both local and server scripts must be parented to specific locations to be able to run. When scripts of any type are parented to a storage service, they will not run until they are moved to an area where they can. This behavior could be compared to the `Disabled` property of a script going from `true` to `false`.

When assets are contained within `ServerStorage`, they can only be accessed by the server. This behavior makes `ServerStorage` the best place for storing any instances you don't wish clients to have access to. Furthermore, physical instances from our **Workspace** that are stored in `ServerStorage` will not produce either memory or a rendering load on the client. This provides a greater performance increase than storing something in `ReplicatedStorage`, though the use cases for this service are different in most circumstances.

## StarterGui

The `StarterGui` service is a hub for a multitude of both core and custom user interface functionalities. **Graphical user interfaces (GUIs)** can be created by you, with the service acting as a container and your UI-associated instances being visible on screen when parented to the service in Studio, as shown in the following screenshot. When running the experience, your UI is copied from here and parented to the `PlayerGui` container under a client. More information on UI and its direct scripting implications will be covered in *Chapter 6, Creating a Battle Royale Game*.



Figure 4.7: Parenting GUI instances to `StarterGui` makes them visible on-screen while in Studio

`StarterGui` is also used as an interface to interact with and manipulate core GUI features. In the following example, the service can be used to completely disable a portion of the core UI using the `SetCoreGuiEnabled()` method. To use it, simply specify which UI you want to be affected and provide a bool representing whether that UI should be enabled or disabled:

```
local starterGui = game:GetService("StarterGui")
starterGui:SetCoreGuiEnabled(Enum.CoreGuiType.PlayerList, false)
```

## StarterPack and StarterPlayer

The `StarterPack` and `StarterPlayer` services are a convenient way of defining base behaviors for players and what they spawn with. `StarterPlayer` allows you to define default properties for the `Humanoid` instances used as player characters by changing the properties of the service itself, as shown in the following screenshot.



It should be noted that when you're changing the default behavior under the **World** tab of the **Game Settings** menu, these are the values that are physically changed:

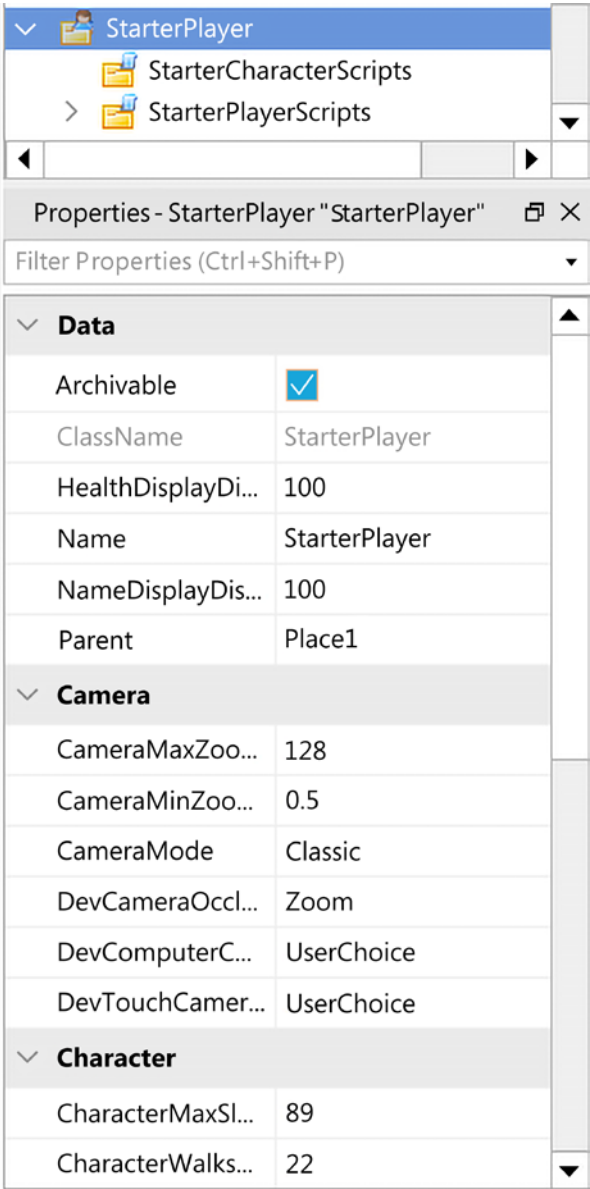


Figure 4.8: Changing the properties under StarterPlayer will change the default player behavior

Additionally, you can store scripts in two locations with different behaviors. Parenting scripts to `StarterPlayerScripts` will cause that script to run from the time the player joins the experience without ever resetting. This makes it the optimal place to put a **local handler**, a local script that manages all subsequent modules; we will cover physical script infrastructure more in the next chapter. When either local or server scripts are parented to `StarterCharacterScripts`, those scripts will be placed directly in a player's character model each time they spawn.

The `StarterCharacterScripts` and `StarterPlayerScripts` instances do not have any additional methods or properties.

In terms of `StarterPack`, anything that's parented here is put in the player's `Backpack` instance once they spawn. Tools are some of the most common items that are put here when you want players to spawn with them by default. Any Tools you wish to add later can be put directly into the `Backpack` under the player.

Like `StarterCharacterScripts`, `StarterPack` resets its contents whenever a player respawns, meaning that if you want a `LocalScript` to be executed each time the player spawns, you can parent that script to `StarterPack`. `StarterPack` itself does not have any unique properties or methods.

## PolicyService

`PolicyService` is a service that allows you to check whether individual users are allowed to see certain content within your experience. For example, depending on the country a user is in or the user's age, they may not be able to spend Roblox or other in-experience currency on random item generators or be able to view links or references to social media. While you may not add this sort of content to your experience, it is worth mentioning should you wish to implement any systems affected by different policies. You can find more information on `PolicyService` here: <https://developer.roblox.com/en-us/api-reference/class/PolicyService>.

## PhysicsService

`PhysicsService` is a way of managing collisions between objects in your experience via the use of **collision groups**. In your experience, you can have up to 32 collision groups and determine how these groups interact with each other. It should be noted that creating, deleting, or modifying the relations between different collisions groups can only be done from server scripts.

In the following code, we are making a collision group for player characters so that players cannot collide with each other. This may be a convenient feature of an **obby** (obstacle course) or any other type of experience where players could be negatively affected by running into others. Collision groups can be created by using the `CreateCollisionGroup()` method of `PhysicsService`. This method simply takes a unique string that can be easily referenced for operations later. Next, we do not want players to be able to collide with each other. The parts of all the player characters will be in one collision group, meaning that we can set the group so that it can't collide with itself. To do this, we can use the `CollisionGroupSetCollidable()` method, which takes two collision groups and a Boolean value to determine whether those two groups can collide. Next, by creating a function using the `player.CharacterAdded` event, which is often used within a `PlayerAdded` event function, a loop is used to iterate over all of the `BasePart` instances within the character. Once found, the parts are assigned to the `Players` collision group using the `SetPartCollisionGroup()` method, which takes a `BasePart` instance and a collision group name as its arguments:

```
local physics = game:GetService("PhysicsService")
physics:CreateCollisionGroup("Players")
physics:CollisionGroupSetCollidable("Players", "Players", false)

player.CharacterAdded:Connect(function(char)
    for _, part in pairs(char:GetDescendants()) do
        if part:IsA("BasePart") then
            physics:SetPartCollisionGroup(part, "Players")
        end
    end

    print(player.Name.. " added to group!")
end)
```

## UserInputService

`UserInputService` is a service that's used in local scripts to detect physical input from a client. It can be used to read input from a player's mouse, keyboard, or any other input device, as well as to determine certain input behaviors.

Let's say that you are looking to detect keyboard input from the client. You can check if a player is even using a keyboard by viewing the value of the `UIS.KeyboardEnabled` property. To detect input of any type, you can use the `InputBegan` event; it specifies what the input was and if the user was interacting with any user interface when that input occurred. In the following example, we are creating a simple interact function that's linked to pressing *E* on a player's keyboard. We do not want this function to run if the client simply pressed the interact key while writing a chat message, so we will immediately return if they were engaging with any UI elements. Next, since we are looking for keyboard input, we can check the `KeyCode` enum (enumeration) of the input and compare it to the desired character using the `KeyCode` enum:

```
local UIS = game:GetService("UserInputService")
UIS.InputBegan:Connect(function(input, typing)
    if typing then return end

    if input.KeyCode == Enum.KeyCode.E then
        print("Client pressed E!")
    end
end)
```

While most services that are practical for a beginner to learn have been covered, you can visit the developer website to learn about what others exist and learn more information regarding their applications. Once on the page linked below, search for service using your browser's **Find** tool to find the services among the other classes listed. Remember that if you are looking for something specific, you can always use the search bar of the developer website: <https://developer.roblox.com/en-us/api-reference/index>.

Services are a core component of creating any Roblox experience and now that you have learned how to utilize them, you can create much more complex systems or even apply lower-level events and methods that would otherwise be inaccessible without defining the service you wish to use. In future chapters, when you build your own experiences, you will use these services in various applications, from player management to experience mechanics.

## Working with physics

Depending on whether you are working as a backend or frontend programmer for a project, essentially meaning you're working with the code infrastructure or with what players see respectively, working with physics may be something you do on a daily basis. While physics directly correlates with math and a backend programmer may work with `CFrame`s and math much more intensely, a frontend programmer needs to work with physical forces in the **Workspace**. There are several ways to do this; that is, primarily with **properties** and **constraints**.

### Constraints

The **Constraint** base class consists of instances that can be utilized to create different physical behaviors within your experiences. Most constraints require `Attachment` instances in order to work. These `Attachments` must be parented to a `BasePart` instance and serve mainly as a single point in a space that constraints can reference.

To make the process of setting up constraints easier, you can use the **Constraints** submenu under the **Model** tab of Studio. This menu allows you to create new constraints without needing to create the attachments for them yourself. Additionally, it includes a convenient drop-down list of all constraint instances, as shown in the following screenshot. Furthermore, you can also view constraints in the **Workspace**, even for those that are normally invisible, by toggling the **Constraint Details** option:

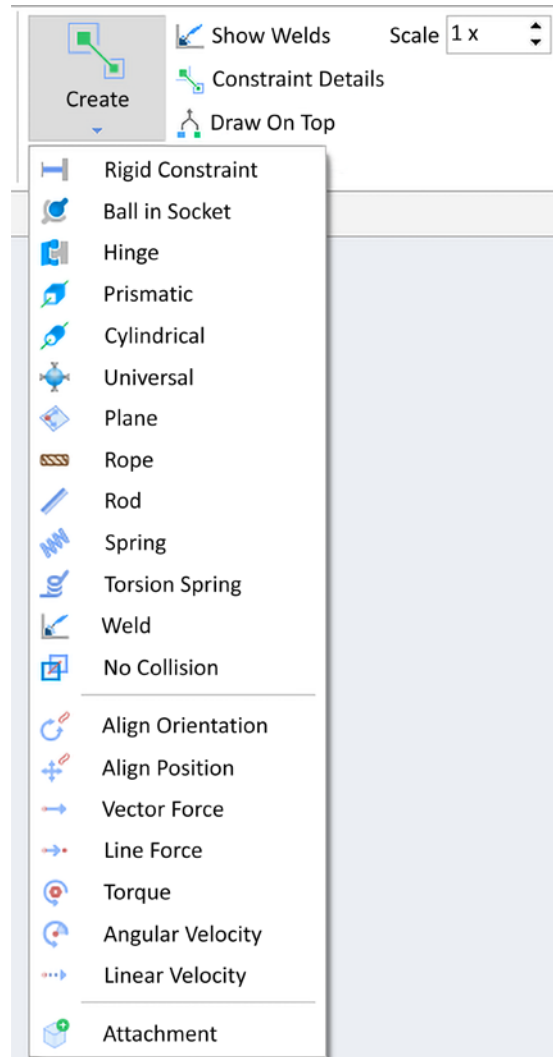


Figure 4.9: There are a variety of constraints that can be used to create unique physical behaviors

The **Rod** constraint is used for keeping parts a set distance from each other, keeping them aligned with a rigid axis created from the two Attachments. Though these parts are kept aligned with each other, they can still freely rotate around their Attachments. The two primary properties of this instance are **Thickness** and **Length**, with **Thickness** being the visual thickness of the constraint and **Length** being the distance between the two parts. You can also view the current distance between the attachments, and subsequently the parts, by checking the **CurrentDistance** property, which cannot be written to.

The **Rope** constraint is an instance that's used to simulate a rope-like connection between two parts. Rope constraints are similar to Rod constraints in that they share the same properties, though Rope constraints have a dampening property called **Restitution**, which is used to adjust how aggressively the rope resumes its original length once force is exerted upon it. To help you conceptualize this constraint better, it is much like a Rod but lacks rigidity – such is the nature of a rope.

In Roblox, **welding** is a way of making parts stay connected while they are unanchored. Several types of instances can be used for making welds, but older weld types break when any of the parts are moved individually with the Studio building tools. They also alter the CFrame of parts when a new weld is created between them. While some of these effects can be negated with code, it is sometimes easier to use a **Weld** constraint instance. By using a **Weld** constraint, you can weld parts together to keep their original positions and have free range of moving the parts once they have been welded, without the connection between them breaking. By toggling **Show Welds** under the **Model** tab, you can view connections like this in Studio, as shown in the following image:

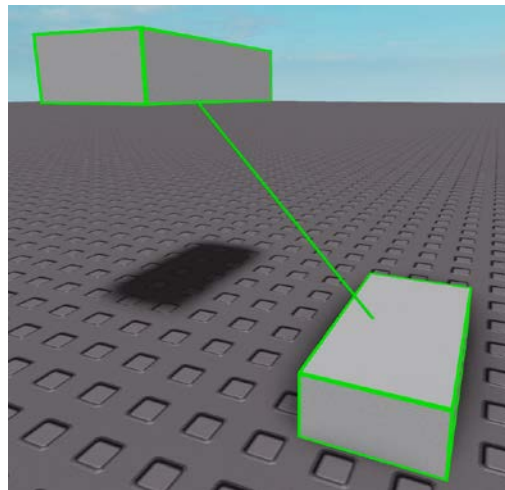
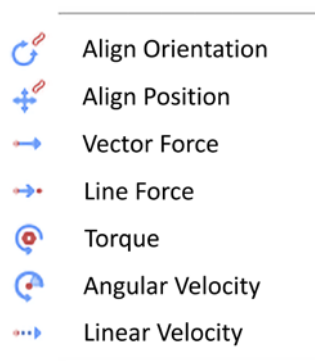


Figure 4.10: You can view connections such as this **WeldConstraint** in Studio by toggling **Show Welds** and **Constraint Details**

To learn more about what other types of constraints there are and how to use them, please take a look at the following article on the developer website: <https://developer.roblox.com/en-us/articles/Constraints>.

## Movement constraints

In the past, instances of the `BodyMover` base class were a convenient way of consistently applying forces directly or applying forces to accomplish a goal when working with `BasePart` instances. However, they have been replaced with new constraints that have all of the previous functionality and more. These constraints require an `Attachment` instance to be parented to the `BasePart` you wanted to affect. This means that if you wanted an entire model to move, the parts would need to be welded together with the constraint in a central part:



*Figure 4.11: There are several constraints focused on body mover movement, each with different behaviors*

`VectorForce` is a constraint that applies a force to an `Attachment` instance. This force is applied relative to the attachment's `Orientation` property by default, but can do so on a world space basis (meaning that it is not relative to the orientation of the part itself) via the `RelativeTo` property, as seen in the code example below. The primary property of this instance is `Force`, a `Vector3` that is applied to the attachment by default but can be applied to the center of mass of the part via the `ApplyAtCenterOfMass` property.

Though applying direct force has many practical examples, in the following code, we will make a part experience zero gravity. To do this, we need an understanding of some simple physics and the `GetMass()` method of `BasePart` instances.



The force that must be used to suspend the part in the air is its mass multiplied by gravity, applied upward on the *y* axis. The `GetMass()` method will give us the part's mass and we can get the current gravity from the **Workspace**'s Gravity property. From this, we can set the Force property of `VectorForce` to `Vector3` containing our computed force on the *y* axis:

```
local part = workspace.Part
local attachment = Instance.new("Attachment")
attachment.Parent = part
local vectorForce = Instance.new("VectorForce")
vectorForce.Parent = part
vectorForce.Attachment0 = attachment
vectorForce.RelativeTo = Enum.ActuatorRelativeTo.World

local requiredForce = part:GetMass() * workspace.Gravity
vectorForce.Force = Vector3.new(0,requiredForce,0)
```

`LinearVelocity` is another constraint that similarly operates on the world space or relative to its attachment via its properties. This instance is particularly useful when you want to set an object's velocity to be consistent without the need for a loop. The two main properties of this constraint are `MaxForce` and `VectorVelocity`. `MaxForce` is a float property and represents the amount of force that is applied to a part when a velocity is set. This means that if the constraint has an insufficient `MaxForce`, that part will be unable to move. The `VectorVelocity` property is a `Vector3` and simply needs to be set to the desired velocity.

In the following example, we have a part moving forward relative to its front face. Notice that we set the `MaxForce` property to `math.huge`, an essentially infinite value; this is so that the mover will be able to influence the part, regardless of the part's mass. As your skills become more advanced, you may want to adjust the value that's used for this property to prevent any unwanted behaviors, but for the examples shown here, this is fine. If you're curious, try finding a sufficient value based on the part's mass, like we did for the Vector force. By multiplying the desired speed by the forward-facing directional vector of the part, or the `LookVector` component of its `CFrame`, we can create a world space velocity that is still moving forward relative to the front face of the part. It is recommended that you run this code in the Command Bar area, or even have it in a server script and test it on Run mode for easier observation:

```
local part1 = workspace.Part1
local attachment = Instance.new("Attachment")
attachment.Parent = part1
```

```

local linearVelocity = Instance.new("LinearVelocity")
linearVelocity.Parent = part1
linearVelocity.Attachment0 = attachment
linearVelocity.RelativeTo = Enum.ActuatorRelativeTo.World
linearVelocity.MaxForce = math.huge
local projectileVel = 5
local vectorVel = part1.CFrame.LookVector * projectileVel
linearVelocity.VectorVelocity = vectorVel

```

While you could set the part's CFrame to look at a target and always move it forward using the previous example, you can create a velocity aimed at a target without manipulating the part directly. Using the previously configured constraint, we can calculate a new directional vector, with its origin at the part the constraint's attachment is currently parented to, and its target being the position of another part. In the following code, we are calculating the new vector by doing a simple calculation. By subtracting the first position vector from the second, we are given a new directional vector where the first position looks toward the second. Note that the target part, Part2, should be anchored for this example. You may also notice the use of Unit with LookVector. We are doing this to normalize the vector so that it has a magnitude of 1, which is important. The vector components of CFrames will always have a magnitude of 1; notice how differently the constraint behaves without the use of a normalized vector for computing our VectorVelocity. You can simply replace every line after projectileVel in the previous example with the following code:

```

local part2 = workspace.Part2
local lookVector = (part2.Position - part1.Position).Unit
--p2 - p1 = directional vector at p1 looking at p2
local vectorVel = lookVector * projectileVel
linearVelocity.VectorVelocity = vectorVel

```

The AlignPosition instance works slightly differently than the previously listed constraints. AlignPosition has four primary properties, these being Position, MaxForce, MaxVelocity, and Responsiveness. The Responsiveness and MaxVelocity properties are used to determine how aggressively a goal is reached and the maximum speed used in trying to reach that goal. The MaxForce property behaves in the same way as it does with the previous constraints we've covered. Position is a Vector3 property and represents where the constraint is attempting to move the part, calculating and applying the necessary forces to meet that goal automatically. In the following example, MaxForce has again been set to math.huge and the Position property has been set to the origin of the **Workspace**, plus 15 studs on the y axis.

Note that this constraint actually has several behaviors and to achieve the one described, we set the `Mode` and `ReactionForceEnabled` properties accordingly. You should see the part smoothly transition between its original position and its new goal:

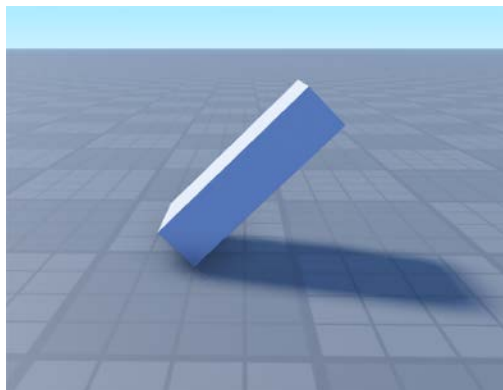
```
local part = workspace.Part
local attachment = Instance.new("Attachment")
attachment.Parent = part

local alignPosition = Instance.new("AlignPosition")
alignPosition.Parent = part
alignPosition.Attachment0 = attachment
alignPosition.Mode = Enum.PositionAlignmentMode.OneAttachment
alignPosition.ReactionForceEnabled = true
alignPosition.MaxForce = math.huge
alignPosition.Position = Vector3.new(0,15,0)
```

The `AlignOrientation` instance is a way of making a part match the orientation component of a provided `CFrame`. The two properties of interest regarding this instance are `MaxTorque`, `CFrame`, and `MaxAngularVelocity`. `MaxTorque` is like `MaxForce` in some of the previously covered constraints; if the constraint does not have a sufficient `MaxTorque` value, then it will not be able to affect the part. `MaxAngularVelocity` is a convenient property that can allow you to specify the maximum rate at which the constraint's goal is met. In the following example, the part will transition from its current orientation to its orientation when turned 45 degrees ( $\pi/4$  radians) around the z axis. Notice that the `MaxTorque` property is once again set to an infinite value so that it can manipulate a part of any mass and that we again set the `Mode` and `ReactionForceEnabled` properties to get our desired behavior:

```
local part = workspace.Part
local attachment = Instance.new("Attachment")
attachment.Parent = part
local alignOrientation = Instance.new("AlignOrientation")
alignOrientation.Parent = part
alignOrientation.Attachment0 = attachment
alignOrientation.Mode = Enum.OrientationAlignmentMode.OneAttachment
alignOrientation.ReactionTorqueEnabled = true
alignOrientation.MaxTorque = math.huge
alignOrientation.CFrame = part.CFrame * CFrame.Angles(0,0, math.pi/4)
```

As a result of our code, our part now sits at a 45-degree angle around its z axis as seen in *Figure 4.12*:



*Figure 4.12: AlignOrientation makes a BasePart meet a set orientation goal*

Now that you have learned how to interact with physics in Roblox outside of a purely mathematical environment, you can create even more complex experience mechanics and visual effects within your experience to provide a better experience for your audience.

## Adding peripheral experience aspects

For your experience to be complete, you must be able to work with the more peripheral aspects of a project. Having the ability to manage sometimes overlooked project details such as sound, lighting, and other effects will make you more experienced and consequently more valuable as a member of a team.

### Sound

In game development, sound is a very important aspect of making your players feel completely captivated by whatever is occurring in a scene or portion of gameplay. While proper sound design is its own subject, you must know where to find sounds and how to manipulate them inside Roblox.

There are several routes you can take to acquire the right sounds, those being searching for audio in the **Toolbox**, looking for models in the **Toolbox** that may contain the sound you need, or uploading your own audio files to Roblox. The first option can be done through the **Toolbox** or Roblox library web page. Using the **Toolbox** or library, you can simply navigate to the audio page and input a keyword; then, the most relevant results will be shown to you. When searching for music or sounds this way, it may be helpful to know that most of the results will be audio uploaded by Roblox. Roblox has licensing agreements with the music companies *APM* and *Monstercat* to provide music to developers without additional costs or attributions.

If you are looking specifically for sound effects, you can use the audio length tool in the **Toolbox** area to find sounds meeting your length criteria, as shown in the following screenshot:

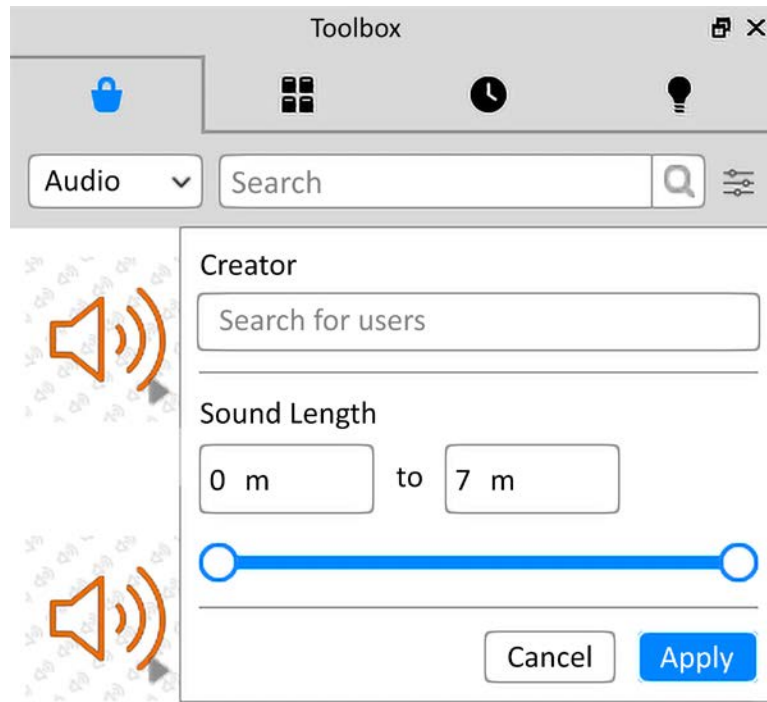


Figure 4.13: Filters make it easier for you to find what you are looking for in the Toolbox

For the second option, set your search to look through models and look for models that are likely to contain which sounds you are looking for. Sometimes, this is more helpful than looking for audio directly because often, audio clips are not named in a way that makes them easy to discover. Lastly, you can upload your own audio assets to Roblox for a small fee, which is dependent on the length of the track.



You need to be aware that, when uploading audio, the sounds you upload are still subject to copyright laws. For example, uploading your favorite song from a certain band would make you liable to receive moderation action against your account if Roblox receives a DMCA from the copyright holder. Roblox does, however, have a few systems in place to protect you. When uploading audio, a bot checks for copyrighted material and will block the content from being uploaded if any is detected. If you still manage to get the sound uploaded and a DMCA is filed, Roblox employs a three-strike copyright offense system. Upon your third failure to observe copyright laws, your account may be subject to suspension or termination. So, while it is perfectly fine to upload audio, make sure you either hold copyright over what you are uploading or have been given the appropriate license to use that audio in your experience.

In late 2016, Roblox released a long-anticipated sound update that allowed for real-time audio effects. Sound modifiers are a convenient way of changing sound behavior without the need to alter sounds in external software and reupload them.

Using these effects may make your environments or experience system more immersive; for example, making use of an echo modifier in a tunnel or changing the pitch of an engine when accelerating:

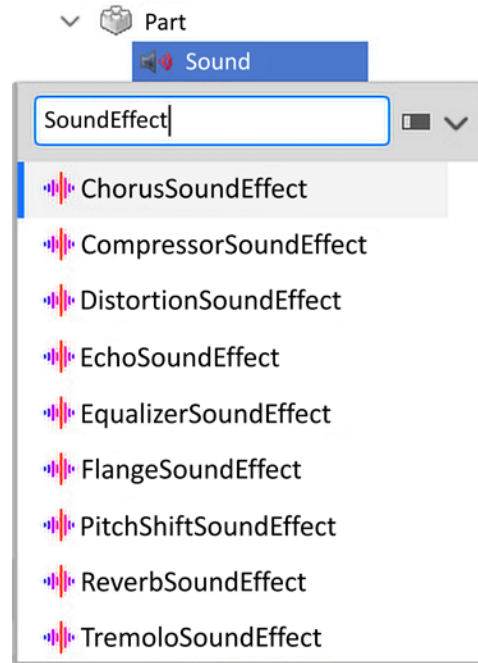


Figure 4.14: Roblox provides sound modifiers for changing sound properties in real time

For more information about the different behaviors of sound modifiers, please read this Roblox blog post from when they were introduced. The post details the effect each instance has and their various properties: <https://blog.roblox.com/2016/11/1/>.

## Lighting

Much like sound, lighting is equally as important when creating an attractive environment within your experience. When working on a team, a builder may be able to fulfill your lighting needs, but knowing how to configure lighting so that it's attractive in your experiences is a good skill to possess, regardless of the role you are trying to fulfill.

The first way you can configure lighting for your experience is via the properties of the **Lighting** service, which is visible under the **Explorer** window. Some of these properties can be seen in the following screenshot:

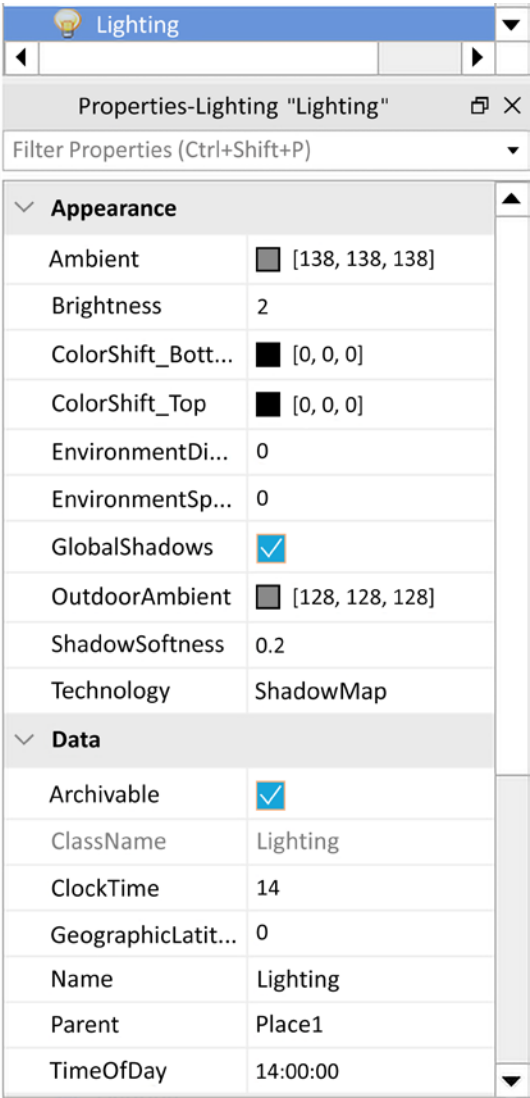


Figure 4.15: Changing the *Lighting* properties from their defaults brings more character to experiences

One of the most used properties of **Lighting** is **ClockTime**. This property ranges from 0 to 24 and represents the hours within the day. It can be easily modified in a loop to create day and night cycles.



The **Technology** section of **Lighting** allows you to choose which type of lighting technology is used in your experience. The current types of lighting technologies are **Compatibility**, **Voxel**, **ShadowMap**, and **Future**. Each lighting technology serves to render lighting in your experience in a different way. Let's take a look:

- The **Compatibility** option renders light in a close approximation of the **Legacy** lighting style, which was removed in favor of the current **Future** lighting technology.
- The **Voxel** lighting option renders light in a similar way to **Compatibility**, but uses a 4x4x4 (voxel) grid for shadow calculations. This makes it more performant than some other lighting technologies, as shadows are less defined.
- The **ShadowMap** option renders shadows with high resolution. However, it is more performant than the **Future** lighting technology because it does not produce accurate reflections in most circumstances.
- **Future** is the newest lighting technology option and supports advanced, high-detail shadows, as well as more realistic reflections on surfaces, from a variety of sources. The only con of this technology is that rendering in those additional visual details takes up more resources, lowering the frame rate and overall performance on weaker devices.

To increase the customizability and variety of aesthetics your experience can have, Roblox has added a number of post-processing effects and lighting modifiers. From sun rays and depth of field, to general color correction, you can design the lighting in your world exactly how you want to better convey the ambience of your experience to players:

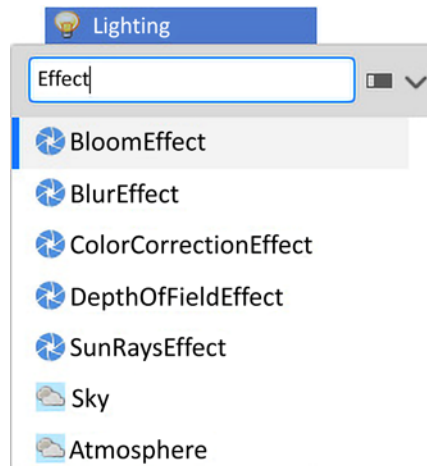


Figure 4.16: Roblox provides lighting effects so that you can easily change the ambience of your experience

By visiting the developer website, you can view some implementations of the aforementioned lighting effects in a detailed environment. Tips and tricks are also included in terms of their best use cases: <https://developer.roblox.com/en-us/articles/post-processing-effects>.

## Other effects

Outside of lighting and sound, there are more direct visual effects that you can use to make your experiences more attractive and interactive. Instances such as **ParticleEmitters**, **Beams**, and **Trails** can be added to create highly customizable effects that can further make your work come to life.

A **ParticleEmitter** is a way of emitting 2D images to make effects such as smoke, fire, and even confetti. These emitters have many properties that can be used to change the appearance of your effects, including particle size, texture, transparency, speed, and particle spawn rate. These emitters even support value ranges that change these properties over the lifetime of the particle. When setting the values of some of these properties from a script, certain data types are used that you have not been introduced to yet, including the **NumberRange** and **NumberSequence** user data. To see some of the best uses of particle emitters, please take a look at the following article on the developer website: <https://developer.roblox.com/en-us/articles/Particle-Emitters>.

**Beams** are a way of making a constant visual effect between two **Attachments**. Additionally, you can add curvature to the path of the effect to make anything from arcing electricity to rainbows. You can change the color, transparency, texture, width, and even texture movement speed from the properties of this instance. In the following image, you can see an arcing dashed line, which could be used to provide a preview of a projectile's path in a physics simulation. Notice that the arc is perfectly aligned between the two **Attachments**:

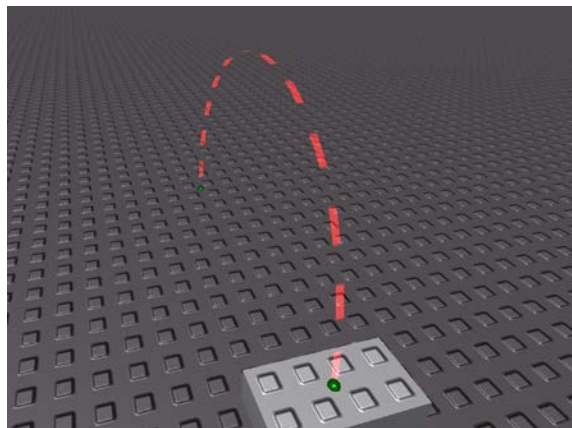
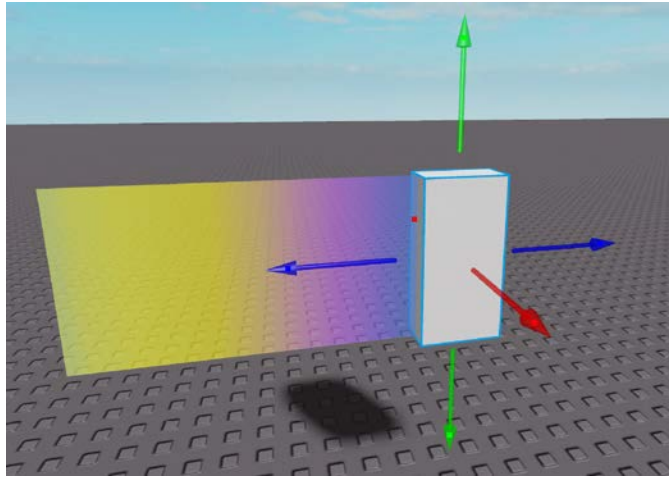


Figure 4.17: Beams can be used to make many effects, including those with curvature

**Trails** are similar to Beams in that they use Attachments, but instead of creating a constantly visible effect between two points, Trails create an effect that is drawn out when the Attachments move and fade away with time. The width of this effect is based on the distance between the two Attachments rather than a property such as Beams. With Trails, you can alter color, transparency, texture, and the time it takes for the effect to fade away. Trails have many uses, including making tracers for projectiles, tire tracks, or just following your character to show movement. Many popular experiences make Trails a collectable or purchasable item, with each one having a unique design or coded perk associated with them:



*Figure 4.18: You can use Trails to indicate the movement of players in your experience's environment*

Knowing how to manage these aspects of experiences may be somewhat peripheral to you as a programmer. However, they are still important parts of the project as a whole, and having the ability to work with these systems is an invaluable skill, regardless of whether you are working solo or in a team.

## Summary

In this chapter, you learned how to use Roblox services to design more complex programs, manipulate Roblox physics to make unique and interactive experience features, and use various effects to enhance your experience. With this, you've begun to master the remaining material needed to make your own full experiences. You will use what you have learned about different script types, `FilteringEnabled`, services, physics manipulation, and additional experience aspects in all your work going forward.

Now that you can program in Luau and know the functionality of various services and instances required to make systems, in the next chapter, you will create your first full experience. You will be given complete guidance to create an **obby**-themed game, a very popular genre on Roblox, that can be **reskinned** to make it unique and capture new audiences.

## Worksheet

- Q1. What is the role of the client and the server in the client-server model?
- Q2. What three script types are primarily used in Roblox development?
- Q3. What can you add at any line in your script to stop the program and examine its state?
- Q4. What does `FilteringEnabled` do?
- Q5. What instances are used for communication between different script instances on the same side of the client-server model?
- Q6. What high-level instances are used for organizing and adding more complex behaviors, additional events, and methods?
- Q7. What services are best for storing assets?
- Q8. What service is used for detecting physical input from the player?
- Q9. What service can have its properties modified to change default character behaviors?
- Q10. What base class of instances is often used for physics manipulation of `BasePart`?



# 5

## Creating an Obby

Now that you have the knowledge needed to make a full experience, you will first put your skills to the test by making an **obstacle course game**, more commonly called an **obby**. From this chapter, you will learn how to integrate features from previous chapters such as moving parts, rewards, effects, and player management. By the end of this chapter, we will have integrated all of the material learned in the previous chapters and tested and published a fully functional experience.

In this chapter, we're going to cover the following main topics:

- Managing player data
- Making obby stages
- Creating rewards
- Adding shops and purchases
- Creating effects
- Testing and publication

### Technical requirements

You will be working solely in Roblox Studio for this chapter. As previously mentioned, you will need an internet connection to use Studio and do independent research on any topics that are covered.

You can find all the code of this chapter in this book's GitHub repository: <https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy/tree/main/Chapter05>.

## Setting up the backend

The **backend** of a system in computing refers primarily to the code infrastructure that a client will never directly interact with. In this section, we will focus on data and player management, which is done from the server. Additionally, as mentioned in *Chapter 3, Introduction to Luau*, for the sake of security, all information that can be obtained by the server should be retrieved by the server.

At the beginning of *Chapter 4, Roblox Programming Scenarios*, you were introduced to modules and some of their possible uses. You will find that when making something larger scale, such as a full project, the modularization of experience systems becomes quite necessary. If you were to use individual scripts instead of modules under one script manager, the organization of your scripts would quickly become a nightmare—something that is not at all manageable for a larger-scale experience. For this chapter, the systems you make will be contained within modules for easy access and editing.

Remember that code inside of modules is simply held and not run unless the module is explicitly required. The following code should be added to a `Script` instance in `ServerScriptService`; the script does not need to be named anything specific but something along the lines of `ServerHandler` is typically good for organization. All of the modules you create should be parented to this script because it serves as a convenient and secure location where clients cannot access them. You should be comfortable enough with programming in a Roblox environment at this point to change the path of something being indexed if you wish to have it located elsewhere. Just remember that `ServerScriptService` is the optimal place to keep backend scripts and modules as they cannot be accessed and subsequently read over by a rogue client.

The code within the `ServerHandler` script is simple—each module acts as an isolated system with the `ServerHandler` script requiring those modules to allow them to run. There still exists a purpose of using modules as opposed to individual scripts, as the `ServerHandler` script will likely be used for wider general management on your own projects and modules will only contain code relevant to an individual system. The following code uses a generic `for` loop to iterate over the modules parented to the script. Remember that the `script` keyword refers to the `Script` instance itself in the **Explorer**. Next, we use `task.spawn()` to make a new thread that envelopes the `require()` function, which takes the current module as its argument.

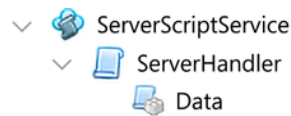
The benefit of making a new thread here is that if a module yields or potentially throws errors when loaded, it will allow other modules to be left unaffected and to load as intended:

```
for _, module in pairs(script:GetChildren()) do
    if module:IsA("ModuleScript") then
        task.spawn(function()
            require(module)
        end)
    end
end
```

Now that we have discussed the structure of the backend, let's see how to manage player data.

## Managing player data

One of the first parts of creating the backend of this experience will be managing player data. Not only will the experience need to keep track of a currency collected by the player, which can be used in a shop, but additionally, we can track player progress to place players where they were during their previous play session. Starting off, we will create a new module called **Data** to contain our code and, as mentioned in the previous section, it should be parented to **ServerHandler**, as shown in *Figure 5.1*:



*Figure 5.1: The Data module is parented directly to the ServerHandler script*

Moving forward, let's learn how to create a datastore system.

## Creating a datastore system

To create the datastore system for this experience, we need to first recognize and define which services will be necessary to set it up. The following code is contained in the **Data** module. You should add the functions seen later in this section to the module after the `dataMod` (a shortening of *data module*) line is declared and before the `return dataMod` line. Note that adding any code after the `return` statement of a module will result in an error.

The **Players** service will be needed so that it can be used in some of the functions that will be added. **DataStoreService** is a service that is used to save data to Roblox servers using an associated key and is not visible in the Explorer.



The **store** variable uses the `GetDataStore()` method of `DataStoreService` to create a new `GlobalDataStore` instance. Data held within this `GlobalDataStore` instance is persistent based on the key it is initialized with; in this case the key is `DataStoreV1`. This means that you can create blank datastores by inputting a key that has not been used in the experience before.

The `sessionData` variable will be used to hold a dictionary containing the data of the players currently in the server using players' `UserIds` as indices and a table of data as the value. A `UserId` is the number that can be found in the URL of a player's profile; it is unique to each user and is not altered by name changes, making it an excellent value to use with data persistence.

Add the following code to your module and we can start creating our experience's data system:

```
local playerService = game:GetService("Players")
local dataService = game:GetService("DataStoreService")
local store = dataService:GetDataStore("DataStoreV1")
local sessionData = {}
local dataMod = {}

return dataMod
```

The first new function we will introduce into our **Data** module is the `recursiveCopy()` function from the *Recursion* section of *Chapter 3, Introduction to Luau*. If you remember from *Chapter 3*, when learning about the table datatype, tables use references, not unique copies, when indexed by variables. By using the `recursiveCopy()` function, you are given some safety when working with data tables in the event that any code you add elsewhere expects a table to be unique:

```
dataMod.recursiveCopy = function(dataTable)
    local tableCopy = {}

    for index, value in pairs(dataTable) do
        if type(value) == "table" then
            value = dataMod.recursiveCopy(value)
        end

        tableCopy[index] = value
    end

    return tableCopy
end
```



Note that you can keep many of these helper functions as local functions within the module, although for a function such as `recursiveCopy()`, it may be advantageous to be able to easily reference it in other scripts. Hence, adding it to the module table would be preferable as any script can require the module to use the function.

For organizational purposes, you may want to practice putting helper functions toward the top of your `ModuleScript`, much like local functions must be in order to be defined wherever they are called from. To completely follow this organization model, I suggest that you put functions you work with the least at the bottom of your script and have core functions located in the top and middle after any needed definitions.

In the next section, we will add the functionalities of creating and loading data for players when they join your experience.

## Creating and loading session data

When a player joins the experience, they need to have their data added to the `sessionData` dictionary. To do this, we will add a new function to our module titled `setupData()` that takes a player instance as its argument. Note that, to retrieve data, we use the `GetAsync()` method of the `GlobalDataStore` instance held by the `store` variable. The value held in the key variable is the `UserId` associated with the player. If the player has stored data, the `data` variable will contain whatever that data is, be it a table or a single value. If no data is associated with the player, then the variable will hold a `nil` value. In the following function, we set up two cases that execute depending on whether there is associated data with the player.

For the first case, we copy a table of stats called `defaultData` and then use a generic for loop to change the values within that copy to match the player's data. The advantage of copying the `defaultData` table as opposed to directly adding the player's data table to `sessionData` is that you can add new values to `defaultData` later, and it will automatically be integrated into the player's current data the next time the player loads, without the need to do any additional coding.

For the second case, nothing needs to be done, as a blank copy of the `defaultData` table has already been assigned to them in `sessionData`. If you so desire, you could make an additional module, located somewhere secure, that holds the `defaultData` table for further cleanliness in your script.

You may notice that a function named `set()` is being called but has not yet been defined. Leave this line in your code, as we will be defining and using it later. You should add the following code into your module without alteration unless you are confident working with this system:

```

local defaultData = {
    Coins = 0;
    Stage = 1;
}
dataMod.load = function(player)
    local key = player.UserId
    local data = store:GetAsync(key)
    return data
end
dataMod.setupData = function(player)
    local key = player.UserId
    local data = dataMod.load(player)

    sessionData[key] = dataMod.recursiveCopy(defaultData)

    if data then
        for index, value in pairs(data) do
            dataMod.set()
            print(index, value)
            dataMod.set(player, index, value)
        end

        print(player.Name.. "'s data has been loaded!")
    else
        print(player.Name.. " is a new player!")
    end
end
end

```



In order for your data system to work, you **MUST** enable the **Enable Studio Access to API Services** setting under the **Security** tab of the **Game Settings** menu. If you are unsure where to locate this menu, refer to *Chapter 2, Knowing Your Work Environment*.

Now, you need to be able to call these functions at the appropriate times. In the module, include the following `PlayerAdded` event function, which calls `setupData()`, passing along the player as an argument. See also that we implement the `leaderstats` system seen in *Chapter 4, Roblox Programming Scenarios*, to display the player's current stats. The values in the `leaderstats` folder are set to their associated values in the `defaultData` table by default but, when data is loaded, the `set()` method is called, which updates the values in the `leaderstats` folder, and ensures that you do not need to create any additional cases:

```
playerService.PlayerAdded:Connect(function(player)
    local folder = Instance.new("Folder")
    folder.Name = "leaderstats"
    folder.Parent = player
    local coins = Instance.new("IntValue")
    coins.Name = "Coins"
    coins.Parent = folder
    coins.Value = defaultData.Coins
    local stage = Instance.new("IntValue")
    stage.Name = "Stage"
    stage.Parent = folder
    stage.Value = defaultData.Stage

    dataMod.setupData(player)
end)
```

Now that data for the player has been created and added to `sessionData`, we will implement functions into the module to manipulate it.

## Manipulating session data

Now that the player has joined and their data has been properly set up, you need to be able to manipulate what is in the `sessionData` table with ease from other scripts. Working once again in the `Data` module, some basic functionality that should be introduced are functions that can set and increment data in the table, as well as a function to retrieve data from the table.

These three functions are very straightforward; for the `set()` and `increment()` functions, the three arguments are the player that you want to manipulate, a string of the stat you want to be changed, and the value you want to be used with the function:

- For the `set()` function, the player's stat in their `sessionData` table is set to the value provided.
- In the `increment()` function, the stat is set to itself plus the provided value.
- For the `get()` function, you only need to provide a player instance and the name of the stat that you wish to read.

Note that both the `set()` and `increment()` functions also update the corresponding stat in the `leaderstats` system to the new value:

```
dataMod.set = function(player, stat, value)
    local key = player.UserId
    sessionData[key][stat] = value

    local statVal = player.leaderstats:FindFirstChild(stat)
    if statVal then    statVal.Value = value
    end
end

dataMod.increment = function(player, stat, value)
    local key = player.UserId
    sessionData[key][stat] += value
    local statVal = player.leaderstats:FindFirstChild(stat)
    if statVal then
        statVal.Value = dataMod.get(player, stat)
    end
end

dataMod.get = function(player, stat)
    local key = player.UserId
    return sessionData[key][stat]
end
```

With the player's data now able to be easily changed, the following section will show you how to save this data when the player leaves the experience.

## Saving player data

Saving player data should be rather familiar after learning how to load data. The function to save data will simply be called `save()` and take a player instance as its argument. Like before, the key used to access the player's data table in `sessionData` is the player's `UserId`. Additionally, the key is used with the `SetAsync()` method of the `GlobalDataStore` instance under the `store` variable to save a copy of that data table. `SetAsync()` takes the key as its first argument and the data that needs to be saved as its second:

```
dataMod.save = function(player)
    local key = player.UserId
    local data = dataMod.recursiveCopy(sessionData[key])

    store:SetAsync(key, data)
    print(player.Name.. "'s data has been saved!")
end
```

When the player leaves the experience, their data table must be removed from `sessionData`. If this were not to occur, your experience would consistently have more memory taken up by each player as they join the experience, and their data would be loaded into the table but not removed. By adding the following `removeSessionData()` function, which takes the player whose data should be removed as its argument, the player's `UserId` can be used as a key to set the player's data table in `sessionData` to `nil`:

```
dataMod.removeSessionData = function(player)
    local key = player.UserId
    sessionData[key] = nil
end
```

Like using the `PlayerAdded` event to call the `setupData` function, we will be utilizing the `PlayerRemoving` event to detect when a player leaves to save their data and additionally remove their data table from `sessionData`. See that the event function calls both the `save()` and `removeSessionData()` functions of the module, passing its player instance argument as a parameter. Remember that calling functions from modules is yielding, so there is no worry of the player's data table being removed before it has a chance to be saved:

```
playerService.PlayerRemoving:Connect(function(player)
    dataMod.save(player)
    dataMod.removeSessionData(player)
end)
```

The following section covering the creation of your own datastore system will focus on certain implementations that will help to make sure your datastore behaves how you want it to, add additional functionality, and learn some more peripheral information about the nature of datastores.

## Addressing throttling and edge cases

Roblox's DatastoreService can experience errors that lead to player data loss but by taking some additional steps, we can reduce this risk significantly. To make your system as perfect as possible, you can implement certain methods that will ensure a player keeps their data if some saving attempts fail.

One of the most common causes of data loss is when **throttling** occurs. **Throttling** is essentially what happens when your request to set or retrieve data from Roblox is denied by their servers. This can happen for a variety of reasons; for example, you may be trying to request information, an action is performed too often, or the service is simply having issues at the time of your request.

An efficient way to prevent this occurrence from damaging a player's data is to introduce something called a `pcall()` (short for **protected call**) function. A `pcall()` function is used to wrap code and return whether the code executed successfully and, if not, what error was encountered. When loading and saving player data, we can wrap the `SetAsync()` and `GetAsync()` method calls with `pcall()` functions and simply retry if the previous attempt to save or load data was unsuccessful.

In the following code, we redefine both the `save()` and `load()` functions of the **Data** module with versions that implement `pcall()` functions. Note that for both of the functions, we check whether the success variable is true and, if not, we call the function again. This loop will continue until success occurs. For saving, we can just call the function but for loading, we do need to implement some recursive logic since something is being returned. As you can see, continued failures will keep calling the `load()` function until there is success, returning the data to where the first retry attempt was made. Make sure to fully replace the two current functions in your module with this code:

```
dataMod.save = function(player)
    local key = player.UserId
    local data = dataMod.recursiveCopy(sessionData[key])

    local success, err = pcall(function()
        store:SetAsync(key, data)
    end)
```

```

    if success then
        print(player.Name.. "'s data has been saved!")
    else
        dataMod.save(player)
    end
end
dataMod.load = function(player)
    local key = player.UserId
    local data
    local success, err = pcall(function()
        data = store.GetAsync(key)
    end)

    if not success then
        data = dataMod.load(player)
    end
    return data
end

```



You can also record the number of retries your program has made with a differently formatted function and kick the player if a limit you set is reached, in an effort to preserve their data, but that will not be necessary for this project.

By creating an autosave system, you can make sure a client's data saves after every set period of time, lessening data loss if their data does not save when leaving the experience. The following function uses a constant time defined in the `AUTOSAVE_INTERVAL` variable, which should be put toward the top of your module.

Using a while loop, a generic for loop iterates through the `sessionData` table, obtaining the player from their `UserId`, which is subsequently their data table key, by using the `GetPlayerByUserId()` method of the `Players` service. Once the player has been found, the `save()` function of the module is called, using the player as a parameter. It is important that you do not make the value held by the `AUTOSAVE_INTERVAL` too small, as saving too frequently may increase the risk of throttling:

```

local AUTOSAVE_INTERVAL = 120
local function autoSave()
    while task.wait(AUTOSAVE_INTERVAL) do

```



```

    print("Auto-saving data for all players")
    for key, dataTable in pairs(sessionData) do
        local player = playerService:GetPlayerByUserId(key)
        dataMod.save(player)
    end
end
end
spawn(autoSave) --Initialize autosave loop

```

Another measure you can take to protect player data is by adding a `BindToClose()` function to your **Data** module. The `Close` event of game fires when a server instance shuts down because all players have left, or the experience has been manually shut down by the developer. When this occurs, the `PlayerRemoving` event of the `Players` service can be inconsistent in firing. The following function will not only make a request to save every player's data by way of the `dataMod.save()` function but it will also prevent the server from closing for up to 30 seconds until the save requests have been fulfilled:

```

game:BindToClose(function()
    for _, player in pairs(playerService:GetPlayers()) do
        dataMod.save(player)
        player:Kick("Shutting down game. All data saved.")
    end
end)

```

With the datastore system now created, you are encouraged to add whatever stats you want to the `defaultData` table and ensure that the system works by using the testing features of Studio. One way to do this is to change your stats in the `PlayerAdded` event function after your data has loaded, test, leave, and rejoin to make sure you have the stats you left the testing session with, rather than the values contained within the `defaultData` table.

The following section will introduce you to some physical mechanics that should be implemented into your obby.

## Managing collisions and player characters

Since players could potentially be negatively affected by colliding with other players while navigating the stages of your obby, you will want to introduce a collision group so that collisions between players are not possible, like in *Figure 5.2*:



Figure 5.2: With collision groups, we can make it so players can't collide with each other

To do this, you should create a new module named `Physics`, parented to the `ServerHandler` script like before. The services we will need to define are the `Players` service and `PhysicsService`, as shown in the following code:

```
local playerService = game:GetService("Players")
local physicsService = game:GetService("PhysicsService")
local physicsMod = {}

return physicsMod
```

Next, you do not necessarily need to make any functions within the module for the base of the experience, but you may want to make some later if you were to introduce your own systems. Like in the *Working with physics* section of *Chapter 4, Roblox Programming Scenarios*, we need to create a new collision group using a unique string identifier with the `CreateCollisionGroup()` method of `PhysicsService`.

Following that, the `CollisionGroupSetCollidable()` method of `PhysicsService` is used to make `BasePart` instances within the collision group not collide with each other. This allows us to make a `PlayerAdded` event function, which creates a new `CharacterAdded` event function for any client that connects to the experience.

A generic `for` loop can then be used to iterate over the `BasePart` instances of the player's character, which is conveniently passed as an argument of the `CharacterAdded` event function. Once a `BasePart` instance is found, it is added to the `Players` collision group by way of the `SetPartCollisionGroup()` method of `PhysicsService`. Players will no longer be able to bump each other off the map or stand on top of each other to get to places they should not be able to, as a result of the following code:

```
physicsService:CreateCollisionGroup("Players")
physicsService:CollisionGroupSetCollidable("Players", "Players", false)
playerService.PlayerAdded:Connect(function(player)
    player.CharacterAdded:Connect(function(char)
        for _, part in pairs(char:GetDescendants()) do
            if part:IsA("BasePart") then
                physicsService:SetPartCollisionGroup(part, "Players")
            end
        end
    end)
end)
```

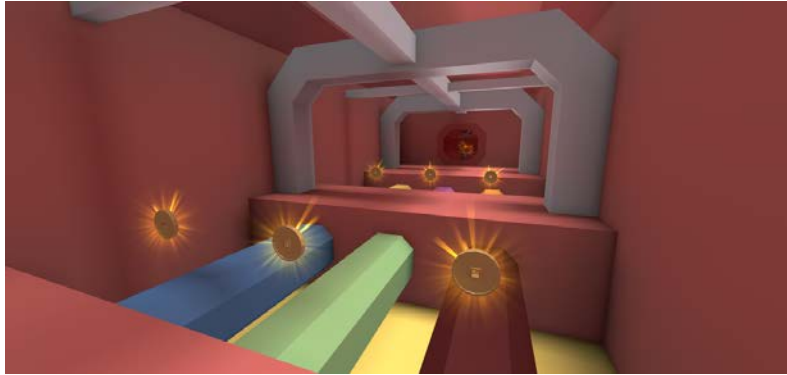
You should keep in mind that collision groups can be used outside of necessary experience behaviors. You can use them to make special stages that make use of collision groups for unique puzzle mechanics. For example, you can make a stage where a player can pass through a wall without any resistance, but a box or ball cannot pass, sort of like some of the puzzle stages seen in the popular *Portal* video game series.

In this section, you learned how to create a comprehensive datastore system that is easily editable and keeps the data of players as secure as possible. Additionally, by way of `PhysicsService`, we made it so players cannot interfere with each other when completing obby stages.

## Making obby stages

Now we get to the most important part of developing an obby: designing your stages! While building and designing your stages is up to you, this section will guide you on how to program various mechanics that can be implemented into your stages to make your experience engaging and profitable.

By the end of this section, you should be able to design stages like the one seen in *Figure 5.3*:



*Figure 5.3: An example of a complete obby stage*

Let's begin by creating part behaviors.

## Creating part behaviors

Once again, the modularization of this system will be critical. Many new programmers who do not take advantage of modules attempt to use individual scripts to manipulate what is sometimes hundreds of parts. This almost immediately leads to headaches if any changes to these scripts need to be made as the code will need to be changed everywhere.

By using a module, the code affecting these parts is isolated to a single location and can be edited easily. Our new module, which should be titled `PartFunctions`, will only need to define a few services or modules since we are primarily working with methods from the `BasePart` base class, as seen in the following code. Additionally, like with our `Data` module, we'll make a table associated with each player within a table. This table is named `coinCodes` as we will use it to keep track of what coins a player has collected:

```
local playerService = game:GetService("Players")
local replicatedStorage = game:GetService("ReplicatedStorage")
local dataMod = require(script.Parent.Data)
local partFunctionsMod = {}
local coinCodes = {}

playerService.PlayerAdded:Connect(function(player)
    coinCodes[player.UserId] = {}
end)
```

```

playerService.PlayerRemoving:Connect(function(player)
    coinCodes[player.UserId] = nil
end)

return partFunctionsMod

```

For these following functions, we will be making use of the Touched event of BasePart instances. If you remember, BasePart instances with a Touched event return whatever other BasePart instance they were hit by.

To limit the amount of redundant code, we will make a helper function named `playerFromHit()` that takes a single BasePart instance as its argument and looks to see whether the part is a descendant of a player's character. If it is, both the player and the player's character are returned in a tuple format; otherwise, both values will be nil by default.

Note that the `FindFirstAncestorOfClass()` method of instances will recursively search for an instance matching the specified class type until a matching instance is found or the experience's DataModel instance is reached; the DataModel is the instance referenced with the game keyword and is at the top of the Roblox parent-child hierarchy. Regardless of whether the Model instance is found or the value is nil, it is passed to the `GetPlayerFromCharacter()` method of the Players service.

This method can take a nil value or any instance as input, but it will return nil unless the instance passed to the method is the character model of a player currently connected to the experience. If there is an associated player with the passed model, then the player is returned by the `playerFromHit()` function, as shown in the following code:

```

partFunctionsMod.playerFromHit = function(hit)
    local char = hit:FindFirstAncestorOfClass("Model")
    local player = playerService:GetPlayerFromCharacter(char)

    return player, char
end

```

As mentioned before, this helper function could be kept as a local function but having it as part of the module allows it to be used in other Scripts that require the module, which may be convenient for a more general behavior like this.

The first function we will introduce to our `PartFunctions` module, called `KillParts()`, is for use with a simple kill part. This is used as an obstacle that a player must avoid when trying to complete an obby stage. When a part with this `Touched` event function linked to it is hit by a player's character, the player will instantly have the `Health` property of the `Humanoid` instance in their character set to 0, killing them. Humanoids exist in all Roblox characters and are used for controlling the behaviors of these character models.

See that the `Touched` event passes a hit part as expected, which can be used as a parameter of a call to the newly implemented `playerFromHit()` method, returning both the player and character if the part is a descendant of a player's character model.

Next, we use short-circuit logic, which was introduced in *Chapter 3, Introduction to Luau*. If the player variable is `nil`, the conditional will stop checking conditions since a requirement of the `and` operator is not met. By ordering our conditions in this way, with the player first and the `Humanoid`'s `Health` second, we avoid using multiple conditional statements or otherwise producing an error. If the character's `Humanoid`'s `Health` is greater than 0, it is set to 0, killing the player, as seen in this code block:

```
partFunctionsMod.KillParts = function(part)
    part.Touched:Connect(function(hit)
        local player, char = partFunctionsMod.playerFromHit(hit)
        if player and char.Humanoid.Health > 0 then
            char.Humanoid.Health = 0
        end
    end)
end
```

While not all that important, the health check is implemented to reduce the number of times any lines of code need to be executed, since the player does not need to be killed if already dead. This is a small optimization that may amount to something valuable if your experience has many stages and players.

The next function is named `DamageParts()` and is used when a part should only damage a player and not kill them instantly. You can place many of these throughout an obby stage to increase the amount of suspense for the player as they try to preserve as much of their health as possible. This function is similar to `KillParts()` but instead of killing the player, it subtracts a specified amount of damage from the player's current health, with the damage being obtained from a number **attribute** of the part named `Damage`.

Attributes are like custom properties which allow you to customize instances with your own data. You can create and modify your own attributes for any instance using many data types. To create an attribute of an instance, scroll to the bottom of the **Properties** menu when an instance is selected in the **Explorer** menu. Here, you should see a button reading **Add Attribute**, as seen in *Figure 5.4*, that, when pressed, brings up a small box where you can enter the attribute name and data type. Once an attribute is added, you can change the value of it from the properties menu:

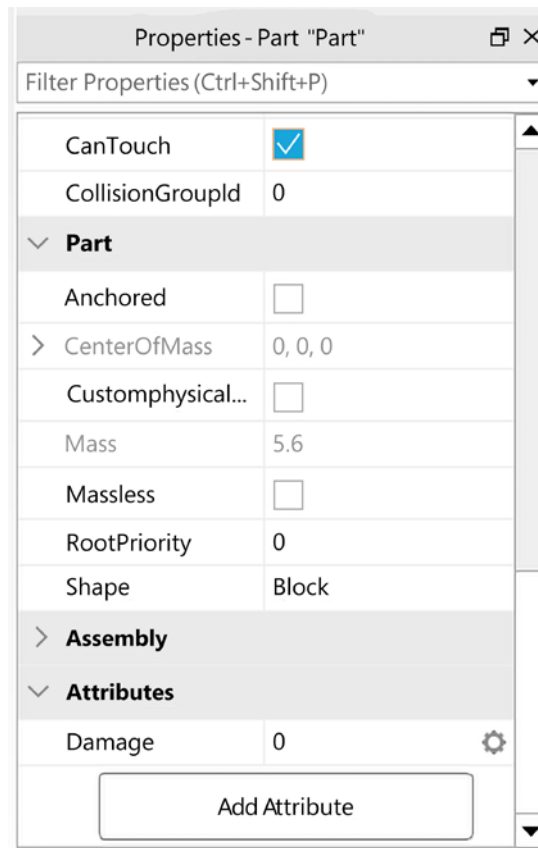


Figure 5.4: Attributes like Damage can be added to any instance, such as a Part

Additionally, we introduce something else new, which is commonly referred to as a **debounce**. A **debounce** is a value that prevents something from running until the value is reset. In the following example, we set the debounce from false to true when the player is damaged and implement a `task.delay()` function, which resets the value after 0.1 seconds.

A `task.delay()` function is created similarly to a `task.defer()` function, but a time argument is used before defining the function. You may wonder why someone would use a `task.delay()` function as opposed to a `task.wait()` function, but a `task.delay()` function has the advantage of not yielding any of the code following it, making it suitable for different use cases.

Since the debounce value is reset after 0.1 seconds pass, the player can only be damaged a maximum of 10 times per second. You will find, when making your own systems, that debounces are important for limiting input or other events from being spammed. Be aware that when using a debounce on the client, someone wanting to exploit the experience can still change the debounce value or avoid it altogether, so do not use it as a replacement for good security if you are communicating with the server:

```
partFunctionsMod.DamageParts = function(part)
    local debounce = false
    local damage = part:GetAttribute("Damage")
    part.Touched:Connect(function(hit)
        local player, char = partFunctionsMod.playerFromHit(hit)
        if player and not debounce then
            debounce = true
            local hum = char:Humanoid()
            hum.Health:TakeDamage(damage)

            task.delay(0.1, function()
                debounce = false
            end)
        end
    end)
end
```

One of the stats included in the player's data is `Stage`, which can be used to save the last stage the player was on and position them whenever they rejoin or respawn. The following function, `SpawnParts()`, will be used with parts that should act as checkpoints, positioned at the beginning of each stage. These spawns should have a number attribute named `Stage` and have the value of them be in increasing order.

Like before, we identify whether a player has been hit and then check whether the current checkpoint has a value one more than the player's current stage, to make sure they cannot accidentally go backward and lose progress or cheat and progress too far forward.



If this condition is met, the player's Stage stat is updated to match the value of the spawn part they encountered:

```
partFunctionsMod.SpawnParts = function(part)
    local stage = part:GetAttribute("Stage")
    part.Touched:Connect(function(hit)
        local player, char = partFunctionsMod.playerFromHit(hit)
        if player and dataMod.get(player, "Stage") == stage - 1 then
            dataMod.set(player, "Stage", stage)
        end
    end)
end
```

While this works as expected, there is no notification that allows the client to feel like they have progressed unless they look at their leaderstats. This will be addressed in the *Creating effects* section of this chapter where we will implement particles and sounds when a player touches a new spawn point.

Now that these functions have been made, you may have noticed that camel case was not used to declare them. This is because you must create Folder instances in the **Workspace** named after each function that has been defined and instances are typically named using Pascal case. This is done so that the following code in the PartFunctions module can easily reference the part folders and call the function with the same name.

Once the folders are added to a table, nested for loops are used to call the function with the same name as the folder, passing along each part within that folder. You must put this code at the end of your module as all of the functions must be defined before they can be called:

```
local partGroups = {
    workspace.KillParts;
    workspace.DamageParts;
    workspace.SpawnParts;
}
for _, group in pairs(partGroups) do
    for _, part in pairs(group:GetChildren()) do
        if part:IsA("BasePart") then
            partFunctionsMod[group.Name](part)
        end
    end
end
```

```

        end
    end
end

```

While all of the obby parts will now work as expected without any additional programming, we must still add functionality to the parts in the `SpawnParts` folder, which currently only update the player's `Stage` stat. To accomplish this, we will create a new module called `Initialize` under the `ServerHandler` script, which will be a place to run more general tasks that are not specific to a particular system when the player spawns.

First, we wait for the `HumanoidRootPart` (the primary part) of the player's character. This is just to make sure that all of their characters have loaded before we continue. You can see that the player's current stage is then obtained from their data, but we need to introduce a new function to get the physical spawn point that corresponds to that number. To do this, we will implement a function named `getStage()` that takes a stage number as its argument. This function will use a generic `for` loop to iterate over all parts within the `SpawnParts` folder, returning a part if its `Stage` attribute matches the provided `stageNum` argument. With the part returned to the main function, the `CFrame` of the `PrimaryPart` of the player's character is set to the `CFrame` of the found spawn location with an offset so that they do not move to the part's exact location, as seen in the following code:

```

local playerService = game:GetService("Players")
local dataMod = require(script.Parent.Data)
local spawnParts = workspace.SpawnParts
local initializeMod = {}
local function getStage(stageNum)
    for _, stagePart in pairs(spawnParts:GetChildren()) do
        if stagePart:GetAttribute("Stage") == stageNum then
            return stagePart
        end
    end
end
end

playerService.PlayerAdded:Connect(function(player)
    player.CharacterAdded:Connect(function(char)
        player:WaitForChild("HumanoidRootPart")
        local stageNum = dataMod.get(player, "Stage")
        local spawnPoint = getStage(stageNum)

```

```

        char:SetPrimaryPartCFrame(spawnPoint.CFrame * CFrame.new(0,3,0))
    end)
end)
return initializeMod

```

Now that the main special parts for your obby have been created, the next step is to keep players engaged by finding multiple ways of rewarding them.

## Creating rewards

While players may enjoy your obby for its challenge, theme, or unique puzzles, you can further engage your audience by adding rewards for reaching checkpoints or taking unnecessary risks in stages. In the player's data, there is a stat called Coins, which we can manipulate to give players a currency they can redeem for in-experience items.

To change a player's Coins stat, we will create a new function named `RewardParts()` in the `PartFunctions` module; make sure that the function is put before the `for` loop that applies functions to parts. Like before, you will need to add a folder with the corresponding name to the `partGroups` table so that parts in the folder can be processed.

The following function will first capture the number of coins that should be awarded from a number attribute of the part named `Reward`. Following this, we will make a unique code for the coin so that we can track which coins a player has collected and prevent them from claiming a coin more than once.

In the `Touched` event function, the player is acquired, and the function looks to see whether a folder of coin codes titled `CoinTags` already exists in the player. If that folder does not exist, then the folder is created. Now that the folder has been created, we make sure no instances in the folder have the code as their name, indicating that the coin has not been collected before. If this check is passed, the reward amount is given to the player and a tag is added to the folder.

You may have noticed that the part is still visible and that there is nothing that really indicates you collected a coin; this will be addressed in the *Creating effects* section of this chapter. Keep in mind that if the player were to rejoin the experience, whether it is the same server or not, the coins will respawn on the stages. If this is not how you want your obby to behave, you can make a constant name or code for coins and save the code in the player's data:

```

local uniqueCode = 0

partFunctionsMod.RewardParts = function(part)

```

```
local reward = part:GetAttribute("Reward")
local code = uniqueCode
uniqueCode += 1

part.Touched:Connect(function(hit)
    local player = partFunctionsMod.playerFromHit(hit)

    if player then
        local playerCoinCodes = coinCodes[player.UserId]

        if not table.find(playerCoinCodes, code) then
            dataMod.increment(player, "Coins", reward)
            table.insert(playerCoinCodes, code)
            replicatedStorage.Effect:FireClient(player, part)
        end
    end
end)
end
```

One way to make players feel more engaged is to create and award them badges. If you want to award badges when a part is touched, you could implement the following function.

We will define this function as `BadgeParts()` to work with our previously made part-processing system and additionally define `BadgeService` at the beginning of the module with the other services. The ID of the badge you want to be awarded should be the value of a number attribute of the part named `BadgeId`. Remember that the `BadgeParts` folder will need to be defined and added to the `partGroups` table. After the `BadgeID` has been obtained, we check for a valid player and continue.

By using the `UserHasBadgeAsync()` method of `BadgeService`, which takes a player's `UserId` and a `BadgeId`, we can check whether the player is already in possession of the badge so that a warning is not produced in the Output window, saying that the badge has already been awarded to them. If you recall from *Chapter 2, Knowing Your Work Environment*, badges can only be awarded to a player once unless they delete them from their inventory.

After checking that the player does not already own the badge, it is awarded to the player by using the `AwardBadge()` method of `BadgeService`, which similarly takes a player's `UserId` and a `BadgeId`, as seen in the following code:

```
local badgeService = game:GetService("BadgeService")
partFunctionsMod.BadgeParts = function(part)
    local badgeId = part:GetAttribute("BadgeId")
    part.Touched:Connect(function(hit)
        local player = partFunctionsMod.playerFromHit(hit)

        if player then
            local key = player.UserId
            local hasBadge = badgeService:UserHasBadgeAsync(key, badgeId)
            if not hasBadge then
                badgeService:AwardBadge(key, badgeId)
            end
        end
    end)
end
```

## Shops and purchases

Now that the stages of your obby are functionally complete, you can begin to monetize it by introducing shops that make use of in-experience currency as well as Robux! Introducing an economy into your experience will not only create a new way to engage your players but also make money with in-experience purchases. You can see an example of a UI-based shop in *Figure 5.5*:



Figure 5.5: An example of a well-designed UI-based shop

## Robux premium purchases

To generate revenue from your experience, you must implement some Robux-only items that your players can purchase. As mentioned in *Chapter 1, Introducing Roblox Development*, Roblox developers primarily make money from selling items in their experiences as well as from Premium Payouts. This section will show you how to make a dynamic sales system as well as some specific powerups for your obby.

For this system, we will be introducing a new module called `Monetization`, parented to the `ServerHandler` script. This module will define two services we have not worked with before, those being `InsertService` and `MarketplaceService`.

`InsertService` is used primarily to load assets from the website from an identifying `assetId`. An asset's `assetId` is typically the numbers seen in the URL of an item on the Roblox website.

`MarketplaceService` is used to manage Roblox purchases, including processing purchase data to comply with Roblox regulations as well as awarding players once they have purchased an item from your experience.

The module will contain the following code to start:

```
local playerService = game:GetService("Players")
local dataService = game:GetService("DataStoreService")
local insertService = game:GetService("InsertService")
local marketService = game:GetService("MarketplaceService")
local dataMod = require(script.Parent.Data)
local monetizationMod = {}

return monetizationMod
```

The first function we'll add to the Monetization module is called `ownsPass()`. This function will take the `UserId` of the player and the ID of the Pass we want to see if they own. Since this function makes requests to the Roblox website via the `UserOwnsGamePassAsync()` method of `MarketplaceService`, we will need to wrap it in a `pcall()` function. Note that this method caches whether the user owns the Pass, so if they don't own the Pass when they join the experience, it will likely always return `false`, even if they purchase it while in the experience. Conversely, if they own the Pass, join the experience, and delete it, the method will likely still return `true`. We'll implement something to track whether a user has purchased a Pass while in the experience in this section:

```
monetizationMod.ownsPass = function(userId, passId)
    local doesOwnPass = false
    local success, _ = pcall(function()
        doesOwnPass = marketService:UserOwnsGamePassAsync(userId, passId)
    end)

    if not success then
        return monetizationMod.ownsPass(userId, passId)
    end
    return doesOwnPass
end
```

The next function we will introduce to the Monetization module will be called `insertItem()`. This function will be used to load and insert Gears from the website using `InsertService` and parent it to the Backpack instance of the desired player. By using the `LoadAsset()` method of `InsertService`, the asset corresponding to the passed ID will be loaded into the experience. Assets inserted with this service will be parented to a model for organization.

This means that our item must be taken out of the model and the model removed from the experience using the `Destroy()` method of instances.

The `Destroy()` method will completely remove an instance from your game and, like the `Clone()` method, it cannot be used with certain protected instances, like services. Once our tool is indexed, it is parented to the player's Backpack instance:

```
monetizationMod.insertTool = function(player, assetId)
    local asset = insertService:LoadAsset(assetId)
    local tool = asset:FindFirstChildOfClass("Tool")
    tool.Parent = player.Backpack
    asset:Destroy()
end
```

With the `insertTool()` function now created, we need to write the code that will appropriately call it. The following code example adds functions to `monetizationMod`, using a `gamepassId` or `developer productId` as the function's index. This format is important as it will allow us to easily call a corresponding function when a purchase is made, using the ID of the item purchased.

Keep in mind that you will have to upload your own Passes to the experience from the **Create** page and replace the zeroes in the indices with the numbers from the Pass's URL. Additionally, we include one currency developer product that can be purchased multiple times by the player:

```
monetizationMod[000000] = function(player)
    --Speed coil
    monetizationMod.insertTool(player, 99119158)
end
monetizationMod[000000] = function(player)
    --Gravity coil
    monetizationMod.insertTool(player, 16688968)
end
monetizationMod[000000] = function(player)
    --Radio
    monetizationMod.insertTool(player, 212641536)
end
monetizationMod[000000] = function(player)
    --100 Coins
    dataMod.increment(player, "Coins", 100)
end
```



When prompting Passes, we use the `PromptGamePassPurchaseFinished` event of `MarketplaceService` to detect when the player has finished interacting with the prompt. The event passes along as arguments to your function the player who was prompted, the ID of the prompt sent to them, and whether the player bought the item that was prompted to them. If the player did purchase what they were prompted, we can simply call the already defined function, which has an index in the module matching the ID passed by the event.

Note that we also introduce a new service called `CollectionService`. This is used to add tags to instances as well as check for them. The `AddTag()` method of the service takes an instance and a string as its two arguments. You can see that a tag is added to the player with its value as the ID of the Pass purchased; this will be important later in this section:

```
local collectionService = game:GetService("CollectionService")
marketService.PromptGamePassPurchaseFinished:
Connect(function(player, gamePassId, wasPurchased)
    if wasPurchased then
        collectionService:AddTag(player, gamePassId)
        monetizationMod[gamePassId](player)
    end
end)
```

For developer products, Roblox has not made the process as streamlined. As before, the reason our previous example uses a `gamepassId` as the function's index is so that we can make a behavior happen when processing a purchase. For a dev product purchase to correctly process so that you may receive the Robux from a sale, Roblox requires that you utilize a `ProcessReceipt` event function.

This event is part of `MarketplaceService` and is fired whenever a dev product purchase is made within your experience. For these purchases to be recorded, you must make a datastore called `PurchaseHistory`, with the best practice being that you record the player's `UserId` and what they purchased.

The string holding both parts of this data is then used as a key to save a value of `true` in the datastore. Lastly and most importantly, the function returns the `PurchaseGranted` option from the `ProductPurchaseDecision` enumeration. This is the part that tells Roblox the purchase was successful but does not falsely report a successful purchase.

For our following function, the only time a purchase is unsuccessful is when the player no longer exists, meaning they left the experience before the purchase could process, in which case, we return the `NotProcessedYet` option.

The following function should be put into the module without alteration. You should also take note of the `PromptProductPurchaseFinished` event function. This event is fired whenever a developer product prompt is closed, much like the `PromptGamePassPurchaseFinished` event for Passes.

This function calls the function in the module with the corresponding ID of the item being purchased, allowing you to define behavior for when products are purchased, just like Passes. You should add the following code to your module without alteration unless you are aware of what you are doing:

```
local PurchaseHistory = dataService.GetDataStore("PurchaseHistory")
function marketService.ProcessReceipt(receiptInfo)
    local playerProductKey = receiptInfo.PlayerId .. ":" .. receiptInfo.
    PurchaseId
    if PurchaseHistory.GetAsync(playerProductKey) then
        return Enum.ProductPurchaseDecision.PurchaseGranted
    end
    local player = playerService:GetPlayerByUserId(receiptInfo.PlayerId)
    if not player then
        return Enum.ProductPurchaseDecision.NotProcessedYet
    end

    PurchaseHistory:SetAsync(playerProductKey, true)
    return Enum.ProductPurchaseDecision.PurchaseGranted
end
marketService.PromptProductPurchaseFinished:Connect(function(playerId,
    productId, wasPurchased)
    if wasPurchased then
        local player = playerService:GetPlayerByUserId(playerId)
        monetizationMod[productId](player)
    end
end)
```

To learn more about why this function must be implemented into your experience, you can read the API reference covering it on the developer website with the following link: <https://developer.roblox.com/en-us/api-reference/callback/MarketplaceService/ProcessReceipt>.

Now that players have been given tools, it is important that you add a new function call within the `CharacterAdded` event function of the `Initialize` module. This function, named `givePremiumTools()`, will look to see whether the player owns any of the Pass tools and will add them to the player's backpack if they do.

By utilizing the `UserOwnsGamePassAsync()` method of `MarketplaceService`, we can see whether a player owns a Pass by providing a `UserId` and `gamepassId`. This method functions differently than you might expect. In 2018, this function replaced the now deprecated `UserHasPass()` method, which yielded for a measurable amount of time so that it could report whether a user did, in fact, own the specified Pass.

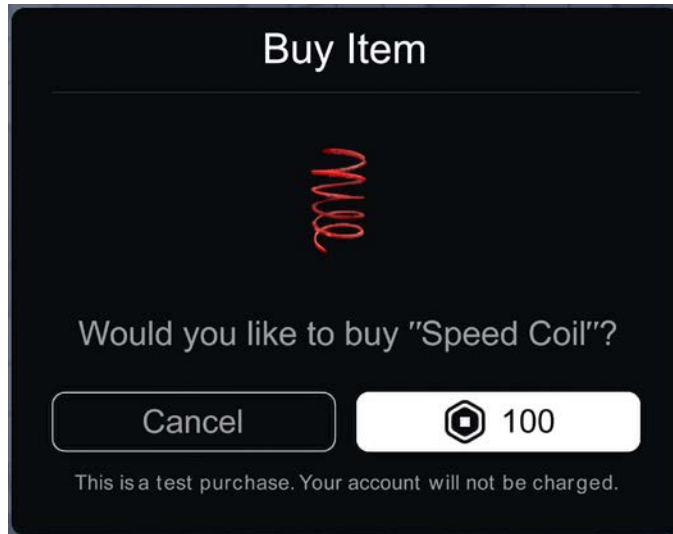
The new method **caches** the result from calling the function, meaning that the function will return the same result for the whole experience session. Because of this, if a player were to purchase a Pass while in an experience and the new method was called, it would still say that they do not own the Pass.

To address this problem, we will be using temporary tags made in the `PromptGamePassPurchaseFinished` event function as well as the `UserOwnsGamePassAsync()` method to determine whether to give a player tools when they respawn.

Note that we have a table of Pass IDs, which a generic for loop then iterates over, calling the associated function in the `Monetization` module if the player owns the Pass. The code to grant players their premium tools is as follows:

```
local collectionService = game:GetService("CollectionService")
local marketService = game:GetService("MarketplaceService")
local monetization = require(script.Parent.Monetization)
local toolPasses = {000000, 000000, 000000}
--Call this function in the CharacterAdded event function ---of
--the Initialize module to award gamepass tools
initializeMod.givePremiumTools = function(player)
    for _, passId in pairs(toolPasses) do
        local key = player.UserId
        local ownsPass = monetization.ownsPass(key, passId)
        local hasTag = collectionService.HasTag(player, passId)
        if hasTag or ownsPass then
            monetization[passId](player)
        end
    end
end
```

Now that processing purchases has been done, we will introduce a new function into the `PartFunctions` module. This function will be called `PurchaseParts()`, and you will need to introduce a new folder into the **Workspace** of the same name and add it to the `partGroups` table. The purpose of this function is to prompt both Passes and developer products to a player when they touch a part. A typical purchase prompt will look like the interface seen in *Figure 5.6*:



*Figure 5.6: This is the standard interface players see when prompted to make a purchase*

After creating the header for the `PurchaseParts()` function, we create two variables that correspond to two attributes of the parts named `PromptId` and `IsProduct`. The `promptId` variable holds the ID of the Pass or dev product that you want to be prompted when the part is touched. The `isProduct` variable is a Boolean value that determines whether what you are prompting is a Pass or developer product.

This must be specified, as the function used to prompt a purchase changes depending on the type of purchase. Once those two values are assigned to the variables, we set up an if-else case based on the purchase type. If the purchase type is a developer product, then we use the `PromptPurchase()` method of `MarketplaceService`, passing along the player that hit the part and `promptId`.

If the purchasable item is a Pass, then we use the `PromptGamePassPurchase()` method of `MarketplaceService`, once again passing along both the player and `promptId`.

The following function will automatically prompt the Pass or product based on the values within the part:

```
local marketService = game:GetService("MarketplaceService")
partFunctionsMod.PurchaseParts = function(part)
    local promptId = part:GetAttribute("PromptId")
    local isProduct = part:GetAttribute("IsProduct")
    part.Touched:Connect(function(hit)
        local player = partFunctionsMod.playerFromHit(hit)
        if player then
            if isProduct then
                marketService:PromptProductPurchase(player, promptId)
            else
                marketService:PromptGamePassPurchase(player, promptId)
            end
        end
    end)
end
```

## Making in-experience currency shops

Now that players with Robux can make purchases within your experience, you need to create something everyone can take advantage of by making a shop system that uses the freely attainable Coins currency created earlier.

For demonstration, we will make one item, a Tool instance, that can be purchased using Coins. To create your Tool, simply add a new **Tool** into the **Workspace** and add a part named **Handle** to it. This **tool** will be named Spring Potion; you should store this tool and any others that you make under a new folder in ReplicatedStorage named ShopItems. This Spring Potion is designed to only be used once and grant the player that consumes it 30 seconds of higher jumping ability.

Keep in mind that if the player dies, they will not be awarded this tool back. First, add a LocalScript instance into the Tool; it can be titled whatever you want but something like ToolHandler would work well. Next, we must then index the player, their character, the tool itself, and the player's mouse. Note that we also create three variables that hold information about whether the tool is equipped, whether the player has already used the tool, and the value that the JumpHeight property of the Humanoid of the player's character should be set to.

By using the Equipped and Unequipped events of the Tool instances, we can create two functions that change the **equipped** variable to record the correct state of the tool. By using the Button1Down event of the mouse, we can detect when the player makes a click. If the Tool is equipped and a click is detected, then the JumpHeight property of the Humanoid of the player's character will be increased from its default value of 50 to the value under the JUMP\_HEIGHT variable, which is 90. Using a task.delay() function, the program waits 30 seconds before setting the player's JumpHeight property back to its default value and destroying the tool. The code of the ToolHandler is as follows:

```
local playerService = game:GetService("Players")
local player = playerService.LocalPlayer
local char = player.Character or player.CharacterAdded:Wait()
local tool = script.Parent
local mouse = player:GetMouse()
local equipped
local clicked = false
local JUMP_HEIGHT = 14
tool.Equipped:Connect(function()
    equipped = true
end)
tool.Unequipped:Connect(function()
    equipped = false
end)
mouse.Button1Down:Connect(function()
    if equipped and not clicked then
        clicked = true
        char.Humanoid.JumpHeight = JUMP_HEIGHT
        task.delay(30, function()
            char.Humanoid.JumpHeight = 7.2
            tool:Destroy()
        end)
    end
end)
```

With an item to give the player now created, we will return to the PartFunctions module and introduce a new part folder and associated function, both titled ShopParts. Remember that you will need to create a new folder in the **Workspace** for these parts and add the indexed folder to the partGroups table for functionality to be applied to them.

First, we must create a table of items that use the name of the associated Tool as indices and a table holding the price and other information you wish to add about the Tool as a value. Like the `defaultData` table, you can alternatively store this information in a module for further organization. Since `ReplicatedStorage` is not yet defined in the `PartFunctions` module, you will want to make a variable for it. You should now use the same format as before to create the new `ShopParts()` function.

In the `Touched` event function, the name of the item that the player wants to purchase is taken from a `String` attribute of the part called `ItemName`; the name of the item is then used to index the tool's information from the `items` table. Once the player that touched the part has been acquired, we ensure that they have enough money to make the purchase, using the price of the item from the tool's information table and the `get()` method of the `Data` module.

If the player has enough currency to make the purchase, then the price is subtracted from their current amount of coins using the `increment()` method of the `Data` module. After indexing the `ShopItems` folder made previously, we duplicate the Tool by using the `Clone()` method of instances and parent it to the player's backpack. The `Clone()` method creates an exact copy of the provided instance regarding properties; the cloned item will not be parented anywhere, such as in a new instance.

You should be aware that not every type of instance can be cloned, particularly important ones such as services—trying to do so will result in an error. As mentioned in *Chapter 4, Roblox Programming Scenarios*, the `LocalScript` in the Tool instance will now be a descendant of the player's Backpack instance, somewhere it can be executed, meaning that the tool instance's program will now work as expected without any additional input from elsewhere:

```
local items = {
    ["Spring Potion"] = {
        Price = 5;
    };
}

local replicatedStorage = game:GetService("ReplicatedStorage")
partFunctionsMod.ShopParts = function(part)
    local itemName = part:GetAttribute("ItemName")
    local item = items[itemName]

    part.Touched:Connect(function(hit)
```

```
    local player = partFunctionsMod.playerFromHit(hit)
    if player and dataMod.get(player, "Coins") >= item.Price then
        dataMod.increment(player, "Coins", -item.Price)
        local shopFolder = replicatedStorage.ShopItems
        local tool = shopFolder:FindFirstChild(itemName):Clone()
        tool.Parent = player.Backpack
    end
end)
end
```

## Preventing exploits

Though this obby does not have much to be exploited, the best security practices have still been used. Whenever possible, data was acquired from the server as opposed to letting a client return that needed information. Additionally, by tracking a player's collected coins on the server, there is no way that the client can exploit and make a request for more coins.

In the next chapter, *Chapter 6, Creating a Battle Royale Game*, you will be working with systems that must receive information from the client, and, in turn, you will need to employ sanity checks and gain more practice with security overall. For the time being, you may want to come up with your own function that deletes any `BodyMover` or `Constraint` instances that the client adds to their character's `PrimaryPart` to prevent exploiters from flying or otherwise cheating to pass stages.

With these additions, players can now purchase items in your experience with Robux or with the in-experience currency you award them. You are also more aware now of what is good security and how we have actively been implementing it so far. In the following sections, you will create the **frontend** of your experience.

## Setting up the frontend

Your experience is now functionally complete with monetization implemented. The **frontend** of a project refers to the part of the experience that players see and interact with the most. The following steps are to beautify your stages, make your obby come to life with effects and movement, and provide visual and auditory indicators to the client to let them know when something has occurred.



Let's start by seeing how to create effects for your obby.

## Creating effects

When creating any visual effects, you want to be mostly operating on the client. This is because we want most of these effects to only appear locally (only to the client who activated the effect) for the sake of performance and to negate replication, meaning that if you want a sound to play when you collect a coin, you should be the only one hearing that sound.

To do this, you should create a new script under `StarterPlayerScripts` titled `LocalHandler` that contains the same code as the `ServerHandler` script. The structure of systems going forward will be the same, working in modules and organizing code based on the system. Our first step in adding effects to our stages will be to create a new module in the `LocalHandler` script titled `Effects`. You will need to define `ReplicatedStorage` because we will want to make use of server-to-client communication by way of `RemoteEvent` instances:

```
local replicatedStorage = game:GetService("ReplicatedStorage")
local effectsMod = {}

return effectsMod
```

## Sound

With the module setup, we will want to introduce some helper functions for our effects. Our first function will help us to work with sounds. Sound design for an obby is not the most important aspect, but some things you may want to include are a music loop as well as some sound effects for when a player activates a part.

While the music loop can simply be a `Sound` instance in the **Workspace** with its `Playing` and `Looped` properties set to `true`, making something happen on activation will require a little bit more programming. In the following example, the code will allow for a `Sound` instance within the provided part to be played.

A particular place where this may be used well is with the `RewardParts` folder, so that the player may associate a certain sound with receiving a reward. For the following code, we will use the `FindFirstChildOfClass()` method, which is inherited by all instances, to find a `Sound` instance of any name that is parented to the part.

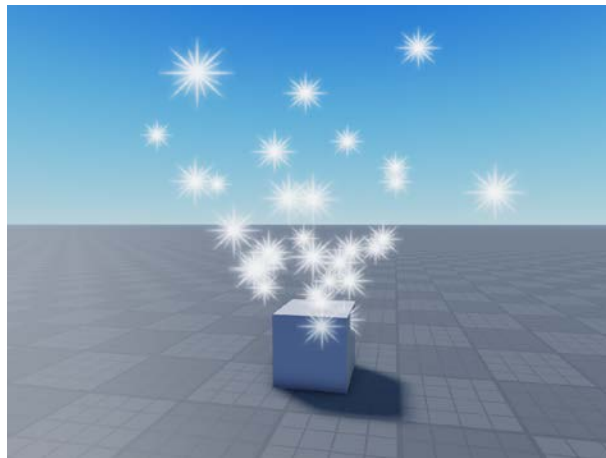
If the sound has been found, then `Sound` is returned and the `Play()` method of `Sound` is called, playing the sound where only the client can hear it, as the call has been made from a module being executed by the client:

```
local function playSound(part)
    local sound = part:FindFirstChildOfClass("Sound")
    if sound then
        sound:Play()
    end

    return sound
end
```

## Particles

Similar to the ability to find and play `Sound` instances, we will want to introduce a function that can emit particles if a `ParticleEmitter` instance is found within the part, as seen in *Figure 5.7*:



*Figure 5.7: Particles can increase the visuals and interactivity of your experience*

Particles were introduced in *Chapter 4, Roblox Programming Scenarios*, and are an easy way to beautify your experience and catch the eye of your players. For the following function, after finding an existent `ParticleEmitter` instance, we use the `Emit()` method, which takes several particles to be emitted.

Be aware that the `ParticleEmitter` instance in your part should be disabled by default when doing this. Moreover, the `Emit()` method will produce these particles all at once, so if you wanted to simply turn on the emitter and turn it off again after some time has passed, you could alternatively set the `Enabled` property of `ParticleEmitter` to `true` and implement a `task.delay()` function to disable it again:

```
local function emitParticles(part, amount)
    local emitter = part:FindFirstChildOfClass("ParticleEmitter")
    if emitter then
        emitter:Emit(amount)
    end
    return emitter
end
```

## Tying in effects

Now, like in the `PartFunctions` module, we will make functions that correspond to a folder name so that they can be easily called. In the `PartFunctions` module, in each part function that you want to add a sound or particle to, you should insert the line that is denoted by `server` after all checks have been passed.

Keep in mind that you will also need a new `RemoteEvent` instance in `ReplicatedStorage` named `Effect` so that the server can send a signal to the client. This line will send a signal to the client, passing along the part that has been hit so that the client does not need to do separate and unnecessary hit detection for every special part in the experience. Once the signal is sent, it will be detected in the client-side `Effects` module.

By making an `OnClientEvent` event function, the folder name of the part's group is obtained from its parent, calling the function with the associated name in the `Effects` module. The functions in the module will be able to use our previously defined helper functions by passing along the part. For parts parented to both the `RewardParts` and `SpawnParts` folders, we check for both `ParticleEmitter` and `Sound` instances using the `emitParticles()` and `playSound()` functions.

Remember that you are not required to include those instances and the function will still work whether they are present or not. For checkpoints in the `SpawnParts` folder, we additionally change the material to neon, changing it back to smooth plastic a second later by way of a `task.delay()` function:

```
--server
--place this in a part function after all checks are made
```

```
replicatedStorage.Effect:FireClient(player, part)

--client
replicatedStorage.Effect.OnClientEvent:Connect(function(part)
    local folderName = part.Parent.Name
    effectsMod[folderName](part)
end)
effectsMod.RewardParts = function(part)
    part.Transparency = 1
    playSound(part)
end
effectsMod.SpawnParts = function(part)
    playSound(part)
    emitParticles(part, 50)
    part.Material = Enum.Material.Neon

    task.delay(1, function()
        part.Material = Enum.Material.SmoothPlastic
    end)
end
```

## Part movement

Another type of effect that can be added to breathe more life into your obby is part movement. While movement can be done on the server, this generally is not good practice as the server refreshes slower and a client may not have a perfect connection to it, resulting in slow and choppy effects. To accomplish this, we will be making a table of parts in the `Effects` module so that parts from any group can be rotated if they contain a value.

To rotate the parts, we will be using a special loop by way of `RunService`. This is used to create loops that run at speeds shorter than a `task.wait()` function can achieve. The `Heartbeat` event fires after each time a frame is rendered. The event passes along a `delta`, the amount of time between each frame, meaning that if a player were getting 60 frames per second, `dt` (short for `delta`) would have a value of  $1/60$ .

After acquiring the folder associated with the part, we look to see whether the part has a `Vector3` attribute named `Rotate`. If the value exists, then the part is added to a table, in which all parts that should be rotated are contained.

The fact this value holds a `Vector3` means that you can give parts custom rotation velocities. As you will see, this value should be the speed you want the part to rotate at in degrees per second.

The `Heartbeat` event function fires after each frame, iterating over all parts by using a generic `for` loop. For each part, the value of the `Rotate` attribute is assigned to a variable called `rot`. This vector is then multiplied by the scalar held in `dt` to convert each component from its original number into a degrees per second format. This new vector is then converted into radians by using another vector-scalar multiplication so that the value may be used with a `CFrame` manipulation of the part.

Lastly, we construct a new `CFrame.Angles()` using the components of the new vector, multiplying it by the parts of the current `CFrame` to achieve rotation:

```
local runService = game:GetService("RunService")
local rotParts = {}
local partGroups = {
    workspace.KillParts;
    workspace.DamageParts;
    workspace.SpawnParts;
    workspace.RewardParts;
    workspace.PurchaseParts;
    workspace.BadgeParts;
    workspace.ShopParts;
}
for _, group in pairs(partGroups) do
    for _, part in pairs(group:GetChildren()) do
        if part:IsA("BasePart") then
            if part:GetAttribute("Rotate") then
                table.insert(rotParts, part)
            end
        end
    end
end
runService.Heartbeat:Connect(function(dt)
    for _, part in pairs(rotParts) do
```

```
    local rot = part:GetAttribute("Rotate")
    rot *= dt
    rot *= ((2 * math.pi) / 360)
    rot = CFrame.Angles(rot.X, rot.Y, rot.Z)
    part.CFrame *= rot
end
end)
```

You may also find that by combining `RunService` with other loops, you can make more effects and mechanics using moving parts including elevators, moving platforms for players to jump between, and other obstacles.

Lastly, lighting is an effect not to overlook. Lighting that is obnoxiously bright or oversaturated is something that could turn away your audience. Many Obbys make the mistake of drastically increasing saturation and brightness to make their stages look more kid-friendly or cartoony but this often leads to an annoying atmosphere for almost any player. While decisions on lighting are ultimately up to you, make sure your map is easy to look at and that you can clearly see the entire stage you are currently on.

Your experience is now more interactive and beautified, which will increase how much you engage the players of your experience. The following sections will discuss the steps of publishing your obby and what we will do moving forward.

## Testing and publication

Now that your experience has been completed on both the frontend and backend, it is important that you test your stages to make sure that they are able to be passed and that all of the special parts within them are functioning as you intended. If you implemented functionality of your own, you should be particularly sure that your new system works. Make sure that your purchases are processing correctly and that you always have the items you purchased to avoid any complaints from players that do not receive what they paid for.

The best way to look for errors is to press `F9` while testing to bring up the **Developer Console**. This console will allow you to look for output, warnings, and errors on both the server and client when in an experience, as seen in *Figure 5.8*.

Keep in mind that only those with permissions to edit the experience will have access to the server-side command bar:

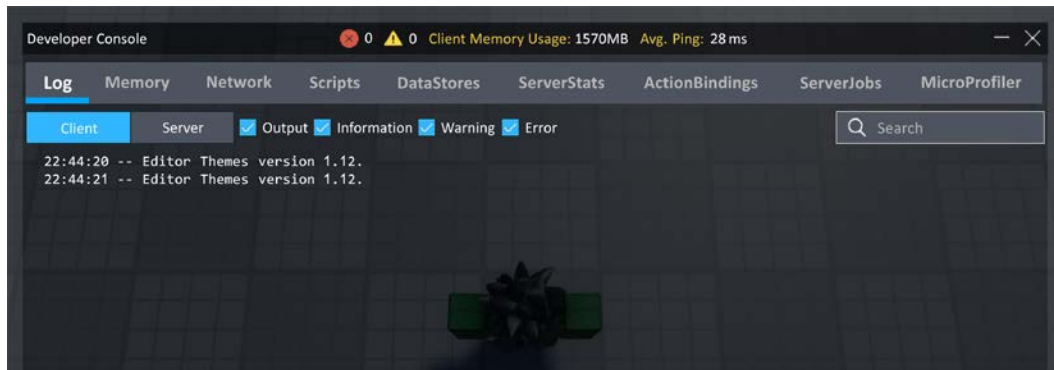


Figure 5.8: The Developer Console serves as a window for output while playing your experience

After testing by yourself, it is a great practice to get feedback from your friends and see whether they find any issues you were not able to, or whether they have any general comments that you want to consider. If you are ready to release your experience, you may find it beneficial to start with a small number of sponsors or ads (see *Chapter 2, Knowing Your Work Environment*, in the *Sponsor this Experience* menu). From this, you can get more feedback from the often all-too-honest public and see what changes they suggest or the further bugs they find; when it comes to finding problems within your experience, there is no such thing as too much testing.

Once your testing concludes and your sponsors have begun running, you have officially released one of your first experiences! There is no better feeling than the mixture of happiness, relief, and sometimes anxiety of publishing a experience. The best thing to do is to let the experience be and monitor its performance. If you have somewhere that players can leave feedback, be it Discord, direct messages, or on the Group Wall if the experience is hosted by a group, see what players have to say and what is worth addressing. Your experience now has the important elements of a good experience: objectives, challenge, and reward. Your players should be able to pass through the different stages of your experience, return later with their progress saved, purchase boosts if they get stuck, and earn badges or currency when they succeed. In terms of design, your stages should be made to have different traps, effects, and, if you're feeling creative, moving components. You can see an example of a well-designed stage in *Figure 5.9*:

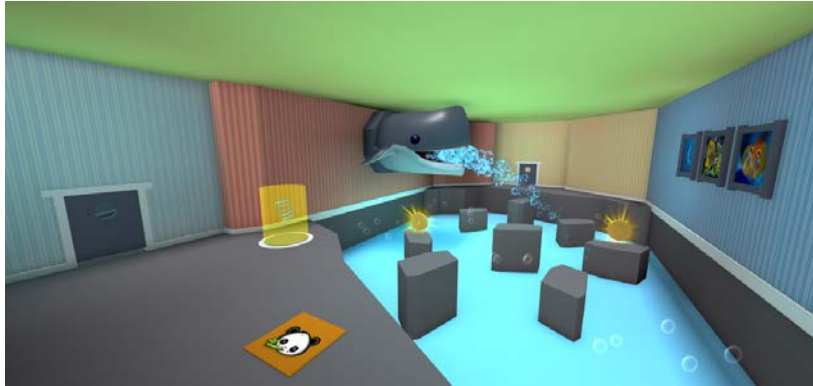


Figure 5.9: An example of a well-designed obby stage

Let's now look at what we've accomplished on the technical side.

## Summary

You have now finished creating what may be one of your first full Roblox experiences! In this chapter, you have learned how to manage players and their data and make systems that add functionality behaviors to your experience as a whole, as well as implement engagement techniques to better attract and keep your audience. Going forward you will make another experience example, which will use some of the systems from this chapter as well as many new ones so that you can implement more complex mechanics into your future experiences.

In the following chapter, you will learn how to create a **Battle Royale**-themed experience. Battle royale games are currently extremely popular, being the concept used in *Fortnite*, *PUBG*, *Call of Duty: Warzone*, and more. An experience of this type will grant you more experience working from the client side and subsequently implementing more security practices, as you will be guided to create weapons, work with the user interface, and generally use more client-to-server communication.

## Worksheet

- Q1. What is the purpose of structuring your experience to be modular?
- Q2. What service is used for storing and retrieving player data?
- Q3. What is the purpose of including a `recursiveCopy` function in the Data module?



Q4. Why do we wrap asynchronous methods in a `pcall()` function?

Q5. What is the importance of including the `ProcessReceipt` event function in your game when selling developer products?

Q6. Why might it be important to spend time adding sounds, particles, and other effects into your experience?

Q7. What method of a `RemoteEvent` can be fired so that the server can communicate with an individual client?

Q8. Why, when using `RunService` loops, might it be important to use the passed `dt` (short for `delta`) value?

Q9. What key can you press while in your experience or play testing in Roblox Studio to see errors from the client or server?

# 6

## Creating a Battle Royale Game

In the previous chapter, you created what may be your first full experience. **Obby** games are a popular concept, in part because of how easy they are to create. In this chapter, we will be making a **Battle Royale** game. This is a popular game genre that includes titles such as *Apex Legends*, *Call Of Duty: Warzone*, *PUBG*, and more. In this game format, players are teleported from a lobby to a battleground where they must find weapons and fight each other, and the last player standing wins the game.

This project will require you to implement all that you have learned from the book so far and learn new material. This material includes working with the **user interface (UI)** and security techniques for systems such as weapons and anything else where the client is communicating with the server.

In this chapter, we're going to cover the following main topics:

- Setting up the backend
- Managing player data
- Setting up the round system
- Creating weapons
- Local replication
- Spawning loot
- Setting up the frontend

## Technical requirements

As with the previous chapter, you will be working solely in Roblox Studio. As before, you will need an internet connection to use Studio. You can find all the code for this chapter in the following GitHub repository: <https://github.com/PacktPublishing/Coding-Roblox-Games-Made-Easy/tree/main/Chapter06>.

## Setting up the backend

Similar to when creating the obby, the backend of the experience should be created to be modular. As before, we will be creating a main server-sided script called `ServerHandler`, which all server modules will be under. So you do not need to refer back to the previous chapter. The code that should be in the `ServerHandler` script is included here:

```
for _, module in pairs(script:GetChildren()) do
    if module:IsA("ModuleScript") then
        task.spawn(function()
            require(module)
        end)
    end
end
```

Moving forward, we will be introducing the modules that should be added to the `ServerHandler` script to create the main functionalities of the experience. As with the previous chapter, you are encouraged to test each system you make, assuming said system is not dependent on one you haven't yet made. Testing throughout the development process not only ensures you do not have bugs but is motivating as well! Next, we will set up data management for players to record their wins and rewards.

## Managing player data

Managing player data will be very easy now that you have already created a reusable datastore system. All you need to do is parent the `Data` module you created in the previous chapter to the `ServerHandler` script. When making experiences of your own design, you can similarly copy and paste this module. Do remember to enable the **Enable Studio Access to API Services** setting in the **Security** tab of the **Game Settings** menu.



If you did not follow the tutorials of the last chapter, you will need to go back to the *Managing player data* section of *Chapter 5, Creating an Obby*, in order to continue with the following steps. Once you have completed that section and its associated parts, you can come back without following any other steps in that chapter.

For this experience, we will need to make some small edits within the Data module pertaining to what data we keep track of. In the last chapter, we only kept track of the player's current stage and the number of coins that they had. For a battle royale game, we will want to keep track of the player's coins as well as how many enemy players they have knocked out and how many rounds they have won. To do this, we will want to change the `defaultData` table to include the `Wins`, `Kills`, and `Coins` stats. Additionally, we will need to create the associated `ValueBase` instances to display on the player's screen using the `leaderstats` system, as illustrated in the following code block:

```
local defaultData = {  
    Coins = 0;  
    Wins = 0;  
    Kills = 0;  
}  
playerService.PlayerAdded:Connect(function(player)  
    local folder = Instance.new("Folder")  
    folder.Name = "leaderstats"  
    folder.Parent = player  
  
    local coins = Instance.new("IntValue")  
    coins.Name = "Coins"  
    coins.Parent = folder  
    coins.Value = defaultData.Coins  
  
    local wins = Instance.new("IntValue")  
    wins.Name = "Wins"  
    wins.Parent = folder  
    wins.Value = defaultData.Wins  
  
    local kills = Instance.new("IntValue")  
    kills.Name = "Kills"
```

```
kills.Parent = folder
kills.Value = defaultData.Kills

dataMod.setupData(player)
end)
```

It is important to note that if you changed the key of your `GlobalDataStore` instance, you may feel free to reset it as data is not persistent throughout different Roblox experiences. Next, we will create the round system of our experience; as mentioned, we need to be able to keep track of which players are alive, award the winner if only one player is left, and then repeat this process.

## Creating weapons

You cannot have a battle royale game without having weapons. In this section, you will learn how to make a comprehensive and secure gun system that makes use of raycasting for hit detection.



Figure 6.1: A laser gun model from the Toolbox

To begin making the weapon system, start by adding a new module to the `ServerHandler` script, named `Weapons`. In this module, we will include the necessary services, variables, and other necessary references. As you can see with the `hitRemote` and `replicateRemote` variables, we will need two `RemoteEvent` instances parented to `ReplicatedStorage`, named `Hit` and `Replicate` respectively. These will be used so that the client and server can communicate with each other when necessary. Implement the following code block into your new module:

```
local playerService = game:GetService("Players")
local replicatedStorage = game:GetService("ReplicatedStorage")
local hitRemote = replicatedStorage.Hit
```

```

local replicateRemote = replicatedStorage.Replicate
local dataMod = require(script.Parent.Data)
local weapons = {}
return weapons

```

You have seen this `playerFromHit()` function in *Chapter 5, Creating an Obby*, in the *Making Obby stages* section. We will be using it in the Weapons module on the server for verifying the shots players make. You can see that it is included as part of the module table rather than as a local helper function, as you may want other systems to be able to freely reference it. You can add this function to the module table without any alterations, as follows:

```

weapons.playerFromHit = function(hit)
    local char = hit:FindFirstAncestorOfClass("Model")
    local player = playerService:GetPlayerFromCharacter(char)
    return player, char
end

```

The next part of this system is to create a new Tool instance within the StarterPack service. In your Tool instance, you should add a new Part named Handle. Due to its name, this Part will automatically serve as the point where a client's character will grip the Tool. Additionally, you should add a new module to the Tool called Settings, which will hold the settings for your weapon. As seen below, some of these settings include how much damage the weapon deals per shot, the rate of fire, the range, and more. The code for this is shown here:

```

local gunSettings = {
    fireMode = "SEMI"; --SEMI or AUTO
    damage = 15;
    headshotMultiplier = 1.5;
    rateOfFire = 300; --Rounds per minute
    range = 500;
    rayColor = Color3.fromRGB(255, 160, 75);
    raySize = Vector2.new(0.25, 0.25); --Width and height
    debrisTime = 0.05;
}
return gunSettings

```

Let's look at what each of these values does to change the appearance and behavior of your weapon, as follows:

- The `fireMode` value is used to determine whether players can continuously fire when holding down their mouse or only fire one shot per mouse click.
- The value held by `damage` is simply subtracted from the health of the player you have hit.
- The value of `headshotMultiplier` is multiplied by `damage` if you shoot an enemy in the head, essentially acting as a *critical* shot.
- The `rateOfFire` value limits how many rounds per minute you can shoot. You can find the minimum amount of time a player must wait between shots by printing `60/rateOfFire`.
- The value of `range` is used to determine how far the projectile you shoot can go; it is set to travel 500 studs by default.
- The `rayColor` and `raySize` values are used to change the color of the projectile visualizer as well as its width and height.
- Lastly, the `debrisTime` value will be used to limit how long a projectile visualizer will be visible for when the weapon is fired.

Now that we have created the base of the `Weapons` module on the server and created the `Settings` module within the `Tool` inside of `StarterPack`, you will need to add a new `LocalScript` titled `ToolHandler`. Since you may want to use this weapon system in different experiences, it will be made as a completely independent system in regard to code so that you do not need to modularize it. Instead, we will be working directly within the script.

In your script, add the following code, which defines the services and instances we will need to reference; you should already be familiar with all of these. In addition to the general definitions, we also create two event functions and a variable to keep track of whether or not the `Tool` is currently equipped, indicating whether the tool is currently usable. The only other declarations you may want to pay attention to are the `firePoint` and `gunSettings` variables. The `gunSettings` variable simply holds the table returned by requiring the `Settings` module within the `Tool`. The `firePoint` variable holds the instance from where your projectile will be shot; this can be the `Handle` `Part` by default since you likely do not have a model for your gun. You should include any particles or sounds you want to occur when your weapon is used in this `Part`, for when the `gunEffects()` helper function is added later in this section. The code for this is shown here:

```
local playerService = game:GetService("Players")
local replicatedStorage = game:GetService("ReplicatedStorage")
local replicateRemote = replicatedStorage.Replicate
```

```

local hitRemote = replicatedStorage.Hit
local player = playerService.LocalPlayer
local char = player.Character or player.CharacterAdded:Wait()
local mouse = player:GetMouse()
local tool = script.Parent
local firePoint = tool:WaitForChild("Handle") --This is where the bullet
comes from
local gunSettings = require(tool:WaitForChild("Settings"))
local equipped = false
tool.Equipped:Connect(function()
    equipped = true
end)
tool.Unequipped:Connect(function()
    equipped = false
end)

```

After implementing the preceding code, add a new Folder called Effects to the **Workspace**. This will be important for the functionality of your weapon's hit detection later in this section.



You may find that depending on your weapon's model, adding an animation of your character holding the weapon will make your experience more aesthetically pleasing. Roblox makes this process very easy to do, and you can learn more about making and using animations in the following article: <https://developer.roblox.com/en-us/articles/using-animations-in-games>.

The first function that we will want to introduce into the ToolHandler script is named `castRay()`. This function will be called to create a new raycast, returning information about which instance was hit, the position it was hit at, the vector direction the raycast was sent along, and the origin from where the raycast was made. The first thing we need to identify is what a raycast really is and what we can use it for. A good way to conceptualize a raycast is as a one-dimensional beam or *ray* that continues from its origin to whatever it hits or, if nothing is hit, continues until it reaches the maximum length of the raycast, which you define in the module via the range value. To elaborate on the concept of *one-dimensional*, a raycast has length but does not have any width or height, meaning that it will always behave like a laser in its hit detection rather than a net.



To create a new raycast, we use the `Raycast()` method of the `Workspace`. This method takes three arguments: an origin vector, a directional vector, and a `RaycastParams` object. We first need a `Vector3` origin of the raycast, as well as a directional vector. Remember that you can get a directional vector by subtracting two position vectors. For example, if you want a directional vector representing position one looking at position two, you could subtract position one from position two ( $P2 - P1 = P1 \text{ looking at } P2$ ). Once these two position vectors have been subtracted to get a directional vector, you will want to get the `Unit` property of the new directional vector. This converts the vector into a unit vector, meaning the vector has a total magnitude of 1. You do not really need to know the meaning behind this, but it is important to do it before multiplying by your scalar range value. With an origin and direction unit vector, we just need to define our `RaycastParams` object. You can create a new `RaycastParams` object via the `RaycastParams.new()` constructor. As with creating a new `Vector3` object, this will need to be assigned to a variable of its own. Once we create the `RaycastParams` object, there are two properties of the object that we want to set, `FilterType` and `FilterDescendantsInstances`. The `FilterType` determines whether we want to only be capable of hitting the instances and their descendants within a table or be able to hit anything but them. These two `FilterType` options are called `Whitelist` and `Blacklist` respectively. This table of instances, regardless of the `FilterType` used, is assigned to the `FilterDescendantsInstances` property. For example, it would be bad if you were able to shoot your own character or the projectile visualizers of other people. To avoid these being hit, we add the character of the client and the `Effects` folder to the `ignoreList` table. Then, we set the `FilterType` and `FilterDescendantsInstances` properties to `Blacklist` and `ignoreList` respectively.

When an instance in the **Workspace** is hit, five pieces of information are returned by the `Raycast()` function via a `RaycastResult` object, those being the instance or terrain cell that was hit, the position it was hit at, the distance of what was hit from the origin, the normal vector to the surface that was hit, and the material of the `BasePart` instance or terrain cell that was hit. The first two of these will be important for this function. It is pertinent to remember that if no instance or terrain is hit, these values will simply be `nil` except for the intersection point, which will just be at the end of the raycast.

After collecting the first two values returned by the raycast, the `Replicate RemoteEvent` in `ReplicatedStorage` is fired. Here, we pass along the `Tool` instance, origin, and point of intersection. We will go over what this remote event does to replicate the projectile visualizer in the upcoming *Local replication* section. For now, we create a projectile visualizer that is only locally visible.

This is just a regular `Part` that has its `anchored` property set to `true`, set so that it cannot be collided with, given a neon material, and—lastly—given its color, width, and height from the `Settings` module. We get the length or depth of the visualizer from the distance between the origin and the point of intersection. Remember that to find the distance between two position vectors, we simply subtract them and get the magnitude of the resulting vector; alternatively, you can use the `Distance` property of the `RaycastResult` object. Once the size has been set, we set the part to be positioned at the raycast origin, looking at the point of intersection and then offsetting the part by half of its size, to be perfectly between the origin and the point of intersection. Now we have a good visualizer of our raycast and, hence, our “projectile.”

With all of the properties of the `Part` set to the correct values, we finally parent the part to the `Effects` folder in the **Workspace**. Since we want this part to only be temporary and disappear after a very short period, we will use the `Debris` service. This service is most often used via its `AddItem()` method, which takes an instance and a length of time; once that time passes, the passed instance is permanently deleted. Looking at the code, we can see that the visualizer is in the `Folder` for the length of the `debrisTime` value in the `Settings` module before being deleted. The default value is a good amount of time for something such as a projectile to be visible. After this, the important information about our raycast is returned to wherever the function was called from. You should implement the following function into your `ToolHandler` script:

```
local ignoreList = {char, workspace.Effects}
local debris = game.GetService("Debris")

local raycastParams = RaycastParams.new()
raycastParams.FilterType = Enum.RaycastFilterType.Blacklist
raycastParams.FilterDescendantsInstances = ignoreList

local function castRay()
    local origin = firePoint.Position
    local direction = (mouse.Hit.p - firePoint.Position).Unit
    direction *= gunSettings.range

    local raycastResult = workspace.Raycast(origin, direction, raycastParams)
    local hit, pos

    if raycastResult then
```

```

    hit = raycastResult.Instance
    pos = raycastResult.Position
end

replicatedStorage.Replicate:FireServer(tool.Name, origin, pos)
local visual = Instance.new("Part")
local length = (pos - origin).Magnitude
visual.Anchored = true
visual.CanCollide = false
visual.Material = Enum.Material.Neon
visual.Color = gunSettings.rayColor
visual.Size = Vector3.new(gunSettings.raySize.X,
gunSettings.raySize.Y, length)
visual.CFrame = CFrame.new(origin, pos) * CFrame.new(0,0,-length/2)
visual.Parent = workspace.Effects
debris:AddItem(visual, gunSettings.debrisTime)

return hit, pos, direction, origin
end

```

The last helper function you should implement inside the ToolHandler script is called `gunEffects()`, which simply iterates over all of the instances that are parented to the `firePoint` part of your Tool. The loop checks for any `ParticleEmitter` or `Sound` instances, calling the `Emit()` or `Play()` methods in each case respectively. Because of the behaviors enforced by `FilteringEnabled`, you will need to play sounds and emit particles for other users when the shot is replicated to them. We will define our replication functions in the next section. You may alter the following code after implementing it, as you see fit:

```

local function gunEffects()
    for _, effect in pairs(firePoint:GetChildren()) do
        if effect:IsA("ParticleEmitter") then
            effect:Emit(50)
        end

        if effect:IsA("Sound") then
            effect:Play()
        end
    end
end

```

```
end  
end
```

Now that all helper functions have been implemented into the `ToolHandler` script, we will create a function for when the player clicks. Before doing this, however, we need to add a new `BoolValue` instance to the `Tool`, called `Debounce`. This will be used as a debounce value that both the client and server can see and manipulate.

By using the `Button1Down` and `Button1Up` events of the `Mouse` instance, we can set behaviors for when the player clicks and releases. Note that, despite the event names, these events will fire appropriately if a player is on a mobile or console. We will create a new function called `fire()` that will call the helper functions and perform any other actions that are necessary for the weapon to shoot. Depending on the `fireMode` setting of the weapon, this function will be called in a loop with a terminating variable that is manipulated by the state of the `Mouse` instance. We will implement a conditional statement inside this function so that the `Tool` must be equipped, and the `Debounce BoolValue` instance within the `Tool` must be `false`, meaning that the weapon is not in cooldown from its fire rate. If these two conditions are met, we will then set the `Debounce` value to `true` and add a `task.delay()` function that will set it back to `false` after a period of `60/rateOfFire` passes.

Following this, we will call `gunEffects()` and `castRay()`, capturing the information that the latter function returns. Next, we check if any instance was hit; if one was, we find the relative `CFrame` between the object that was hit and the point of intersection between that object and the raycast.

A relative `CFrame` essentially tells you the position and orientation of one `CFrame` relative to another. For example, if a cylinder was exactly 5 studs in front of my character, the `CFrame` of the cylinder relative to my character's is `CFrame.new(0, 0, -5)`. In the figure below, we use this logic in a scenario where a cylinder with a diameter of 2 studs is hit by a raycast. The intersection point of the raycast with the cylinder is on the cylinder's front surface. As such, the `CFrame` of the intersection point, relative to the cylinder, is half of its diameter, or its radius, in front of it, that is, in the negative direction of the `z` axis. A relative `CFrame` is found by multiplying the `CFrame` of a `BasePart` by the inverse of the `CFrame` of the `BasePart` you want to be the origin of the relative system. The inverse of a `CFrame` is obtained by way of the `Inverse()` method of `CFrames`.

You can see a visualization of the previously described relative CFrame scenario in *Figure 6.2*, where we multiply the CFrame of the intersection point by the inverse of the CFrame of the cylinder:

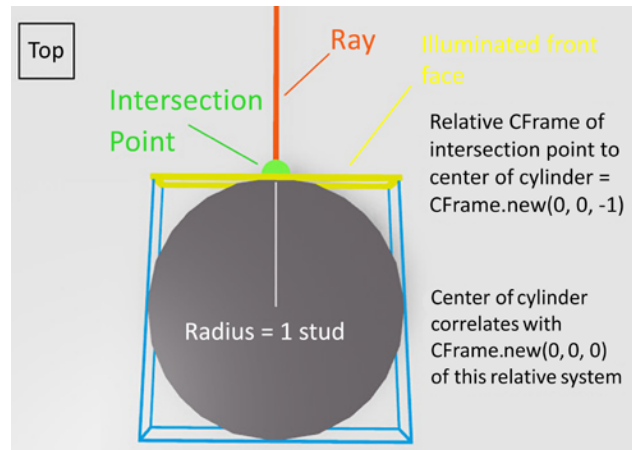


Figure 6.2: This demonstrates the relative CFrame between an intersection point and a cylinder

Once this value has been computed, the `Hit RemoteFunction` in `ReplicatedStorage` is fired, passing along the `Tool`, the hit instance, the direction of the raycast, the origin of the raycast, the relative CFrame of the intersection point and hit instance, and the intersection point in the process. These will all be used for security checks on the server. You should implement the following code into your `ToolHandler` script:

```
local doFire = false
local function fire()
    local waitTime = 60/gunSettings.rateOfFire

    repeat
        if equipped and not tool.Debounce.Value then
            tool.Debounce.Value = true
            task.delay(waitTime, function()
                tool.Debounce.Value = false
            end)
            gunEffects()
            local hit, pos, direction, origin = castRay()
            if hit then
                local relCFrame = hit.CFrame:Inverse() * CFrame.new(pos)
```

```

        hitRemote:FireServer(tool, hit, direction, origin,
                             relCFrame)
    end
end
task.wait(waitTime)
until not equipped or not doFire or gunSettings.fireMode ~= "AUTO"
end
mouse.Button1Down:Connect(function()
    doFire = true
    if char.Humanoid.Health > 0 then
        fire()
    end
end)
mouse.Button1Up:Connect(function()
    doFire = false
end)

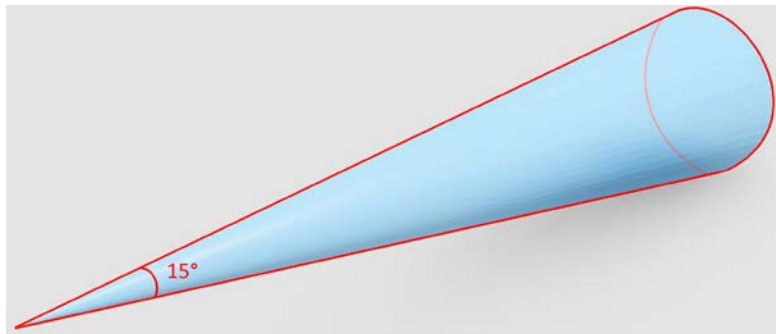
```

Now that the `Hit RemoteEvent` is being fired, we need to define the behavior for it on the server. Working again in the `Weapons` module, we will first need to implement a new helper function called `verifyHit()`, which will be passed the instance that was hit, the direction of the raycast according to the client, the origin of the raycast, the relative `CFrame`, and the settings of the weapon. From this, we will first apply the relative `CFrame` to where the server sees the hit instance to be located. This is a valuable form of protection, as it will soon help us determine whether the client is falsely firing the `RemoteEvent` with mismatching information about the target's location. Once the new `CFrame` has been found, we take the position component of the `CFrame` and create a new directional vector from the server's view of the intersection point and origin. Since the directional vector seen by the server has not had its magnitude changed, taking the magnitude of this vector will give us the distance between the origin and the intersection point. If this distance is greater than the value held by `range` in the weapon's `Settings` module, we know something is not right and the user may be attempting to exploit the system, so we will make an early return, which ends the running of the function.

Next, we will check if the magnitude of either of these directional vectors is zero. The chances of this happening are extremely unlikely, but if either vector has a magnitude of zero, we will not be able to make the next security check. So, we will once again return a `nil` value in this case. Next, we will find the angle between the two vectors by using a dot product between the two vectors and dividing by the product the magnitude of the two vectors.

This will give us the cosine of the angle between the two vectors. Due to floating-point errors mentioned in *Chapter 3, Introduction to Lua*, the number yielded by the two previous calculations may be slightly greater than 1 or slightly less than -1 if the angle is very close to zero or pi radians. This is a problem as the domain of the arccosine function, which we must use to get the actual angle between the two vectors, exists only from -1 to 1, meaning that we would be giving an impossible input to arccosine. To solve this, we can simply implement a conditional statement that says if the cosine of the angle is greater than 1, the actual angle is 0 radians; if the cosine of the angle is less than -1, the actual angle is pi radians. Otherwise, we feed the cosine of the angle to arccosine since it is within the domain of the function. Finally, we convert the angle from radians into degrees, and we officially have the angle between the two vectors in degrees.

We want to effectively make a cone that can contain differences between what the client sees versus what the server observes, to make up for unknown factors and latency. To do this, we will create two new constants in the module called `SECURITY_ANGLE` and `ANGLE_AT_DIST`, which I have made 180 degrees and 3 respectively. In *Figure 6.3*, you can see a demonstration of how the cone is made. See here how the server directional vector and the client directional vector are allowed to be any of the directional vectors contained within and including the perimeter of the cone:



*Figure 6.3: This screenshot demonstrates the security cone and how it infinitely contains many vectors*

After getting the angle, a conditional statement is used to make sure the angle between the two directional vectors is less than or equal to the maximum discrepancy we allow. So, if the server's directional vector is within 180 degrees of the client's reported directional vector at the distance of the intersection, then we conclude that this is a valid hit and return a true value to where the function is called from.

You should implement the following code into your Weapons module, without alteration:

```

local SECURITY_ANGLE = 180
local ANGLE_AT_DIST = 3

local function verifyHit(hit, direction, origin, relCFrame, gunSettings)
    local target = (hit.CFrame * relCFrame).p
    local serverDirection = target - origin

    if serverDirection.Magnitude > gunSettings.range then return end

    if serverDirection.Magnitude == 0 or direction.Magnitude == 0 then
        return end
    local combinedVectors = serverDirection:Dot(direction)
    local angle = combinedVectors/(direction.Magnitude * serverDirection.
Magnitude)
    if angle > 1 then
        angle = 0
    elseif angle < -1 then
        angle = math.pi
    else
        angle = math.acos(angle)
    end
    angle = math.deg(angle)
    local dist = serverDirection.Magnitude
    local adjustedSecurityAngle = math.min(SECURITY_ANGLE, (ANGLE_AT_DIST
* SECURITY_ANGLE)/dist)
    if angle <= adjustedSecurityAngle then
        return true
    end
end
end

```

With the `verifyHit()` helper function created, we can now create the event function for when the `Hit RemoteEvent` is fired. The first thing that occurs when the remote is fired is a call to the `playerFromHit()` function, passing along the intersected instance. Note that in the conditional statement following this call, we check only that `char` exists, which will always be true if the `hit BasePart` is a descendant of a model, regardless of whether there is an associated player.



This is wanted behavior because we check that char exists and that a Humanoid instance exists within it, meaning that you can damage both **non-player characters (NPCs)** and actual players with this gun system since we do not require that a player exists. Additionally, we check that the `Debounce BoolValue` instance within the passed `Tool` is not in cooldown, to ensure exploiters cannot just spam the `RemoteEvent` with requests to damage players. If these conditions are met, then we require the `Settings` module of the passed weapon and call the `verifyHit()` function, which will return `true` if the conditions within it are met.

Assuming the function returns a passing value, the `Debounce` value in the weapon is set to `true` and a `task.delay()` function is created to turn it to `false` after a period of `60/rateOfFire` passes. Lastly, we index the `Humanoid` instance of the hit character model and check if the health of the `Humanoid` is already less than or equal to `0`, meaning we do not need to do anything. If the target is still alive, then we create a new variable to hold the amount of damage that should be subtracted from its health.

If the name of the hit `Part` is `Head`, then we know this was a headshot. We multiply the value held by `headshotMultiplier` by the value of damage and set that as the new value of the damage variable within the event function of the `Hit RemoteEvent`; this damage is then subtracted from the target's health. If the health of the target is now less than or equal to `0`, we know that the player who fired the shot was the one that killed them, and the player's `Kills` stat is subsequently increased. You should add the following function to your module:

```
hitRemote.OnServerEvent:Connect(function(player, weaponName, hit,
direction, origin, relCFrame)
    local otherPlayer, char = weapons.playerFromHit(hit)

    if char and char:FindFirstChildOfClass("Humanoid") and not weapon.
Debounce.Value then
        local weapon = replicatedStorage.Weapons:FindFirstChild("WeaponName")
        local gunSettings = require(weapon.Settings)

        if verifyHit(hit, direction, origin, relCFrame, gunSettings) then
            weapon.Debounce.Value = true
            local waitTime = 60/gunSettings.rateOfFire
            task.delay(waitTime, function()
                weapon.Debounce.Value = false
            end)
```

```

    local hum = char:FindFirstChildOfClass("Humanoid")
    if hum.Health > 0 then
        local damage = gunSettings.damage
        if hit.Name == "Head" then
            damage *= gunSettings.headshotMultiplier
        end

        hum:TakeDamage(damage)

        if hum.Health <= 0 then
            dataMod.increment(player, "Kills", 1)
        end
    end
end
end
end)

```

This section has taught you a great deal about security and working with reliable hit detection methods. In the next section, we will introduce the concept of local replication within your weapons system and how to handle the behavior that should occur when the `Replicate RemoteEvent` is fired.

## Setting up the round system

The main part of creating this experience is setting up the game loop or round system. This backend code will allow you to have an intermission period before each round, add players to the battleground, show everyone who the victor is, and repeat the process.

Before creating this loop, we need to create a lobby for the players to be in during the intermission period or after they die in an ongoing round. This lobby can be designed however you want and can include pads that prompt purchases when touched, like those you created in the previous chapter. Regardless, players will likely spend several minutes in here during their play session, so you want to keep them engaged. To keep players who have already died more entertained, you will be instructed on how to make a **graphical user interface (GUI)** that allows you to spectate other players in the *Working with the UI* section of this chapter.

Additionally, you could make a simple obby with no checkpoints that players can try to complete to gain extra coins while they are waiting in the lobby. You can see an example of a lobby in *Figure 6.4*:



*Figure 6.4: A lobby where players can wait between rounds*

The only thing that your lobby needs to be functional is several `SpawnLocation` instances. If you are unfamiliar with a `SpawnLocation` instance, players spawn at these instances by default when no code directs them elsewhere. Once you add a `SpawnLocation` instance to your lobby, you may want to set the `Duration` property to a low number or `0`. This property controls how long players have a `ForceField` instance protecting their character, but for this experience, there is no point in having one as the only time players shouldn't be able to fight each other is in the lobby, where no weapons are accessible.

The first step of making the round system is to create a new module under the `ServerHandler` script, named `GameRunner`. This module will include the services, variables, and objects needed for the game loop to be made.

You will see that the message and remaining variables hold the paths to two `StringValue` instances named `message` and `remaining`, respectively. These will be used for displaying information to the client during different parts of the game loop. Make sure you add these two `StringValue` instances to `ReplicatedStorage` with the correct names.

Variables that are completely capitalized are called constants for the script. Luau does not have constants as a special construct like other languages do, but when creating a globally defined variable that should never be changed at runtime, each letter should be capitalized, using underscores as spaces, as illustrated in the following code block:

```
local playerService = game:GetService("Players")
local replicatedStorage = game:GetService("ReplicatedStorage")
local dataMod = require(script.Parent.Data)
local random = Random.new()
local message = replicatedStorage.Message
local remaining = replicatedStorage.Remaining
local gameRunner = {}
local competitors = {}
local MIN_PLAYERS = 2
local INTERMISSION_LENGTH = 15
local ROUND_LENGTH = 300
local PRIZE_AMOUNT = 100

return gameRunner
```

Let's go over the meaning of these different constants we have declared in the module, as follows:

- The value held by `MIN_PLAYERS` will be used to determine how many players must be in the experience for it to begin. This value is set to 2 by default as a round with one player would end instantly, allowing players to gain rewards infinitely if they just stayed inside an empty server.
- The second variable, `INTERMISSION_LENGTH`, is used to create a pause between each round once enough players are inside a server. This value should create a long enough wait for people to be able to interact with others in your lobby and rest before starting another round.
- The value held by `ROUND_LENGTH` will be used in a similar manner but utilized only once the round has started. If there has been no victor by the end of the time limit, then the round will end with no winner. This value should allow enough time for players to loot and fight without feeling pressured but not so long that the last two players alive can simply bore everyone who has already died if they are not actively participating in the round.
- The last value, `PRIZE_AMOUNT`, will be used to award coins to whoever the victor of the round is. Feel free to change these values once you have implemented the previous code into the module.

As mentioned in previous chapters, not every helper function within your module needs to be part of the returned module's table if it only has use with a task that is isolated from the module. The following functions, which are used to get and remove players from the `competitors` table, are implemented into the module as local functions since there are not many practical scenarios where they would need to be used by sources other than the module. The `getPlayerInTable()` function is passed a `Player` instance, and the index of the player in the table is returned. This is used with the `removePlayerFromTable()` function, which will be called when a player dies or leaves the experience; this method finds the table index of the player who left the experience or died before removing them from the `competitors` table. You should implement the following code inside the `GameRunner` module, without any changes:

```
local function getPlayerInTable(player)
    for i, competitor in pairs(competitors) do
        if competitor == player then
            return i, player
        end
    end
end

local function removePlayerFromTable(player)
    local index, _ = getPlayerInTable(player)
    if index then
        table.remove(competitors, index)
    end
end

playerService.PlayerRemoving:Connect(function(player)
    removePlayerFromTable(player)
end)
```

## Preparing the player

When the player is about to be deployed to the battleground, there are some things we need to do to prepare them. In the `preparePlayer()` function, the player is given a simple default weapon, which we will learn how to create in the *Creating weapons* section of this chapter, and a new event function is made to handle the case of their death. By way of the `Humanoid` instance of a player's character, we can detect when a player dies by using the `Died` event, and we can subsequently call the `removePlayerFromTable()` function to make sure the player is no longer considered a competitor:

```

local function preparePlayer(player)
    local char = player.Character or player.CharacterAdded:Wait()
    local hum = char:WaitForChild("Humanoid")
    local defaultWeapon = replicatedStorage.Weapons.M1911:Clone()
    defaultWeapon.Parent = player.Backpack

    hum.Died:Connect(function()
        removePlayerFromTable(player)
    end)
end

```

The `addPlayersToTable()` function will be used to add players to the competitors table right when the round begins. By using the `GetPlayers()` method of the `Players` service, we get a table of each player currently connected to the experience and we check that the player we want to add is alive. We do this as we do not want to risk the player being stuck in the lobby if they respawn after the code to move them to the battleground has already been executed. After this condition has been met, the player is added to the competitors table, and they are given their weapon and all other preparation by way of the `preparePlayer()` function. Implement the following function into your module below any functions that it must call:

```

local function addPlayersToTable()
    for _, player in pairs(playerService:GetPlayers()) do
        local char = player.Character or player.CharacterAdded:Wait()
        if char.Humanoid.Health > 0 then
            table.insert(competitors, player)
            preparePlayer(player)
        end
    end
end

```

Next, we will need to bring the players from the lobby to the battleground. To do this, start by adding a new Folder to the **Workspace** named **Spawns**.



It is important that you add at least one spawn point for the number of players that can fit into a server of your experience.



That is to say, if your experience allows for a maximum of 10 players inside a server, there must be at least 10 spawn points. These spawn points should be a regular `Part` instance, not a `SpawnLocation` instance like those in your lobby; if they were `SpawnLocation` instances, players would have the ability to spawn there when they join the experience, like in the lobby.

Once the `Folder` and spawn point `Parts` are created, you should implement the following `spawnPlayers()` function into your `GameRunner` module. This function will create a new table consisting of all spawn `Parts` in the `Spawn` folder and then iterate over all players in the `competitors` table. After identifying a player and that player's character, the `NextInteger()` method of the `Random` object is used to get a random index between 1 and the length of the table. Once the spawn at the corresponding table position has been indexed, the spawn is removed from the table so that other players cannot spawn there, and the player is positioned accordingly. The code for this is shown here:

```
local function spawnPlayers()
    local spawnPoints = workspace.Spawns:GetChildren()

    for _, player in pairs(competitors) do
        local char = player.Character or player.CharacterAdded:Wait()
        local randomIndex = random:NextInteger(1, #spawnPoints)
        local spawnPoint = spawnPoints[randomIndex]
        table.remove(spawnPoints, randomIndex)

        char:SetPrimaryPartCFrame(spawnPoint.CFrame * CFrame.new(0,2,0))
    end
end
```

The last helper function we need to implement for this system is named `loadAllPlayers()`, and it will be called when the round ends, to respawn any remaining players back into the lobby without killing them. This is done by simply iterating over the `competitors` table and calling the `LoadCharacter()` method of the associated `Player` instance.

The code for this is shown here:

```
local function loadAllPlayers()
    for _, player in pairs(competitors) do
        player:LoadCharacter()
    end
end
```

Now that all helper functions have been implemented inside the `GameRunner` module, we must introduce the primary loop that manages the round system in its entirety. We will implement this loop as a function inside the module table so that we can easily use a `spawn()` function with it. This function will be called `gameLoop()`, and by using a `while` loop, it will be able to infinitely run your experience. At the beginning of each loop, it will check that the minimum number of players—as defined by `MIN_PLAYERS`—are in the experience, and then proceed to perform a countdown through the intermission period if that condition is met, changing the message that will be displayed to the players accordingly.

Once the intermission period has concluded, a short period of time is used to tell players to get ready for combat before adding players to the `competitors` table via the `addPlayersToTable()` function. The players added to the table are then spawned at random spawn points by way of the `spawnPlayers()` function. With the players spawned and prepared, the main round clock starts, starting at the value held by `ROUND_LENGTH` and approaching 0; the number of players remaining in the round will also be shown.

You may notice that there are two exit conditions for the repeat loop processing the main countdown. The first condition is if the number of players in the `competitors` table is less than or equal to 1, and the second condition is if the value held by `gameTime` has reached 0. Once the loop exits, a conditional statement makes cases based on which of these conditions are true or false. If the number of competitors is equal to 0 or `gameTime` has become 0, it means that there was no victor as either the last two players killed each other at the same time or the maximum length of the round—as defined by `ROUND_LENGTH`—was reached, again resulting in no victor. Otherwise, the number of competitors must be 1, and the time limit must not have been reached, meaning that we have a victor.

Indexing the victor is easy as they will be the first and only element in the `competitors` table. The winner will have their number of `Wins` incremented by 1 and their `Coins` stat incremented by the value designated in `PRIZE_AMOUNT`. Furthermore, the name of the winner and a short victory message will be displayed for 5 seconds before the entire round begins again.



If you want to experiment, try adding a podium cutscene after each round like *Figure 6.5*:



Figure 6.5: The winner stands victorious on the podium

I encourage that after you implement the following code into your module, you look at the code, following it along closely to understand how this loop accomplished all of the necessary behaviors to make the round system work:

```
gameRunner.gameLoop = function()
    while task.wait(0.5) do
        if #playerService:GetPlayers() < MIN_PLAYERS then
            message.Value = "There must be ".. MIN_PLAYERS.. " players to
                start."
        else
            local intermission = INTERMISSION_LENGTH
            repeat
                message.Value = "Intermission: ".. intermission
                intermission -= 1
                task.wait(1)
            until intermission == 0
            message.Value = "Get ready..."
            task.wait(2)
            addPlayersToTable()
            spawnPlayers()
            local gameTime = ROUND_LENGTH
            repeat
                message.Value = "Time remaining: ".. gameTime
```

```

        remaining.Value = #competitors.. "remaining"
        gameTime -= 1
        task.wait(1)
    until #competitors <= 1 or gameTime == 0
    loadAllPlayers()
    remaining.Value = ""
    if gameTime == 0 or #competitors == 0 then
        message.Value = "There were no victors..."
    else
        local winner = competitors[1]
        dataMod.increment(winner, "Wins", 1)
        dataMod.increment(winner, "Coins", PRIZE_AMOUNT)
        message.Value = winner.Name.." has won the round!"
    end
    competitors = {}
    task.wait(5)
end
end
end
task.spawn(gameRunner.gameLoop)

```

With the main content loop of your game completed, we need to add some elements that will make gameplay the way we want it. In the next section, we will go over how to create weapons so that players can fight each other, making the game end with the best fighter as the champion.

## Local replication

**Local replication** is a term used to describe the process of a client sending a signal to the server, which in turn is sent to other clients to replicate different effects with the benefits of local behaviors. For example, if a client generated a projectile visualizer locally, nobody but that player would be able to see it. If the visualizer were on the server, it would be subject to the latency and lower refresh rate of the server, causing janky visuals. When a client wanting to replicate an effect sends the necessary information to the server so that the effect may be generated for a different user by their own client, then you can maintain accurate and smooth visuals processed at your framerate.

To do this, we will need to create a new `OnServerEvent` event function for the `Replicate RemoteEvent` in `ReplicatedStorage`. This should already be defined from when you created the `Weapons` module earlier. The parameters of this event function are the player who fired the remote, the weapon they used, the origin of their raycast, and the intersection point of their raycast. From these values, we will have the server compute the length and `CFrame` of the visualizer, as well as require the `Settings` module of the weapon, before sending this information to all clients via the `FireAllClients()` method of the `RemoteEvent`. The code for this is shown here:

```
replicateRemote.OnServerEvent:Connect(function(player, weapon, origin,
target)
    local length = (target - origin).Magnitude
    local visualCFrame = CFrame.lookAt(origin, target) *
    CFrame.new(0,0, -length/2)
    local gunSettings = require(weapon.Settings)

    replicatedStorage.Replicate:FireAllClients(player, gunSettings,
    visualCFrame, length)
end)
```

With the signal sent, we must allow every client to receive it and handle the logic for local replication accordingly. To do this, we will add a new `LocalScript` under `StarterPlayerScripts`, titled `LocalHandler`. Since we will have multiple client-side systems you should create this handler to be modular, using the following code, which was also included in the `ServerHandler` script:

```
for _, module in pairs(script:GetChildren()) do,
    if module:IsA("ModuleScript") then
        task.spawn(function()
            require(module)
        end)
    end
end
```

With the `LocalHandler` script now created, the next step is to parent a new module titled `Replication` to it. In the following code, you can see that we define the necessary services and references, as well as the `Replicate RemoteEvent`.

This module only needs to have one main functionality for replication, which is receiving the signal from `RemoteEvent` when all clients are fired. By using the `OnClientEvent` event of the `Replicate RemoteEvent`, we check that the player who used their weapon is not the same player we are trying to replicate to, as this would result in two projectiles. If this condition is met, we can then create a visualizer with the provided information regarding `CFrame` and length while using the `Settings` module from the weapon to apply the correct width, height, color, and time that the visualizer is visible. The code for this is shown here:

```
local playerService = game:GetService("Players")
local replicatedStorage = game:GetService("ReplicatedStorage")
local player = playerService.LocalPlayer
local replication = {}
local replicateRemote = replicatedStorage.Replicate

replicateRemote.OnClientEvent:Connect(function(otherPlayer, gunSettings,
cframe, length)
    if otherPlayer ~= player then
        local visual = Instance.new("Part")
        visual.Anchored = true
        visual.CanCollide = false
        visual.Material = Enum.Material.Neon
        visual.Color = gunSettings.rayColor
        visual.Size = Vector3.new(gunSettings.raySize.X,
gunSettings.raySize.Y, length)
        visual.CFrame = cframe
        visual.Parent = workspace.Effects
        game.Debris:AddItem(visual, gunSettings.debrisTime)
    end
end)
return replication
```

For the next section, you are encouraged to add models to your weapons as opposed to just having a tool handle and fire point. If you want to use more unique models, instead of looking in the Toolbox, try following a Blender tutorial or downloading a weapon asset pack for your experience from places like the Unity Asset Store.

You can see some weapon models I found in the Toolbox in *Figure 6.6*:



*Figure 6.6: Many quality weapon models can be found in the Toolbox*

The Parts of your weapon model will all need to be welded to your Handle Part, as that is where the player grips the Tool. If you are struggling to weld your weapons manually or with your own script in the Command Bar, try using the `qPerfectionWeld` script made by Quenty from the Toolbox or manually welding the model using `WeldConstraints`. This script will automatically weld all the parts within your gun model as long as the script is parented to it. You should create a new Folder in `ReplicatedStorage`, titled `Weapons`, and parent any weapons that you create to this Folder. Remember that you should assign one of the weapons in this Folder to be the default weapon players spawn with in your game loop. Since this will be a default weapon, it should be a weapon that is not very powerful, such as a pistol with a slower rate of fire. Later on, they can find more powerful weapons like rifles or laser guns. All of which you can easily make by editing the weapon's **Settings** module.

With the weapons system now created, we move on to the next section, in which we will learn how to make spawn points that players can find around the map and that have randomly selected weapons to use against enemies.

## Spawning loot

As in most battle royale games, players will start with a mediocre starting weapon and then begin looting around the map for better weapons. In this section, we will be creating a system that spawns weapons around the map for players to find and use.

Keep in mind that for this system, you should have several weapons designed using the weapon system we made previously; the more weapons there are to find, the more players will continue to look for them.

To start this system, add a new module titled `Loot` to the `ServerHandler` script. In this module, add the following code, which defines the services and other references we will need, as well as making the spawn parts hidden, anchored, and made so that you cannot collide with them. As you will see, we will need a new `Folder` called `LootSpawns` that will contain anchored `Parts` that weapons will spawn on. By default, weapon models will spawn one stud above the position of the `Part` itself, so keep this in mind when placing them unless you intend to change this. The following code can be added to the `Loot` module:

```
local replicatedStorage = game:GetService("ReplicatedStorage")
local lootSpawns = workspace.LootSpawns
local weaponFolder = replicatedStorage.Weapons
local random = Random.new()
local weapons = require(script.Parent.Weapons)
local loot = {}

for _, spawnPoint in pairs(lootSpawns:GetChildren()) do
    spawnPoint.CanCollide = false
    spawnPoint.Transparency = 1
end

return loot
```

With the module now set up, we must implement a new function inside the module that spawns one random weapon at each loot spawn location. To prepare for this, let's introduce a helper function called `makeWeaponModel()`. This function takes a `Tool` instance and extracts the `BasePart` instances from it, placing them in a new `Model`. We do not need to keep scripts or any other types of instances, so whatever is left in the `Tool` is destroyed. This new model will serve as the weapon displayed to players on the map. You should now add this helper function to the `Loot` module, as follows:

```
local function makeWeaponModel(weapon)
    local weaponModel = Instance.new("Model")
    for _, child in pairs(weapon:GetDescendants()) do
        if child:IsA("BasePart") then
            child.Parent = weaponModel
        end
    end
end
```

```

        child.Anchored = true
        child:ClearAllChildren()
    end
end

weapon:Destroy()

return weaponModel
end

```

Now, for the main behavior of this module, add a new function named `spawnWeapons()`. This function will use a for loop to iterate over all Parts in the `LootSpawns` Folder and add weapons to be picked up by players at each location. We will be parenting the weapon model to the spawn part, so we will first check that there is no weapon in the Part already and delete it if there is. Next, we will create a pool of weapons and assign the table to the `weaponPool` variable. This pool can be anything; you can change it to contain a different table of weapons if you want certain spawns to be custom, but for this example, we will just use the entire `Weapons` Folder from `ReplicatedStorage`.

Once we create a table of the weapons contained in the Folder, we will create a random index and capture and clone the weapon at the associated table position. Now, we utilize the `makeWeaponModel()` function to create the display model that players will see. With the model now made, we create a new Part that will act as the `PrimaryPart` of the `Model`. This part is made so that we can reposition the `Model` as a whole using the `SetPrimaryPartCFrame()` method of `Model` instances and so that we can use the Part as a hitbox. To make this Part fit for use as a hitbox, we use the `GetBoundingBox()` method of `Model` instances, which returns the `CFrame` description of the `Model` as a whole, as well as the smallest bounding box that contains the `Model`.

Lastly, we will make a new `Touched` event function that checks that a player has touched the `PrimaryPart`. The weapon Tool that is associated with the display model is then parented to the player's backpack, and the weapon model is destroyed so that it cannot be collected again. You should add the following code to the `Loot` module:

```

loot.spawnWeapons = function()
    for _, spawnPoint in pairs(lootSpawns:GetChildren()) do
        local oldModel = spawnPoint:FindFirstChildOfClass("Model")
        if oldModel then
            oldModel:Destroy()
        end
    end
end

```

```
end

local weaponPool = weaponFolder:GetChildren()
local randomIndex = random:NextInteger(1, #weaponPool)
local weapon = weaponPool[randomIndex]:Clone()
local weaponName = weapon.Name
local weaponModel = makeWeaponModel(weapon)
weaponModel.Parent = spawnPoint

local primaryPart = Instance.new("Part")
primaryPart.Anchored = true
primaryPart.CanCollide = false
primaryPart.Transparency = 1
primaryPart.CFrame, primaryPart.Size = weaponModel:GetBoundingBox()
primaryPart.Parent = weaponModel

weaponModel.PrimaryPart = primaryPart
local newCFrame = CFrame.new(spawnPoint.Position) * CFrame.new(0,1,0)
weaponModel:SetPrimaryPartCFrame(newCFrame)

primaryPart.Touched:Connect(function(hit)
    local player, char = weapons.playerFromHit(hit)
    if player and char then
        local tool = weaponFolder:FindFirstChild(weaponName):Clone()
        tool.Parent = player.Backpack
        char.Humanoid:EquipTool(tool)
        weaponModel:Destroy()
    end
end)
end
end
```





With these functions now implemented, it is important that in the `GameRunner` module you define the `Loot` module and add the line `lootMod.spawnWeapons()` at the start of each round, after players are added to the `competitors` table. We do this to make sure new loot spawns at each loot spawn location at the beginning of each round.

If you want to make finding weapons more exciting, you can add sounds or particles to the weapon display model to give them a sort of glow or shine. You can see an example of this in *Figure 6.7*:



*Figure 6.7: Adding particles and other effects to rare weapons makes finding them more exciting*

You should also now have the skills to add a rarity system if you desire. By making a module of weapon names with an associated rarity for each weapon, you can set different values for the chances of the tier of the weapon that spawns. This may not be necessary for your game, but it may make players feel more rewarded when they acquire a weapon that is considered rare.

## Setting up the frontend

With the backend of the experience fully complete, the remainder of this chapter will be used to create what players will interact with most directly. This will primarily include working with the UI.

## Working with the UI

Up until this point, we have stayed away from creating or using any interface. The closest thing we have come to it is working with the `leaderstats` system, which is entirely automated. In this section, we will be creating a simple UI using instances provided by Roblox in order to become familiar with their associated properties, display important information to players, and provide additional engaging features.

## Game message and remaining players display

To create UI for your experience, the first thing that you will need to do is navigate to the StarterGui service under the Explorer. After finding it, you should parent a new ScreenGui instance directly to the service and name it Main. By default, the properties of the ScreenGui instance should not need any changes. These instances serve as containers for the UI that players will actually see and interact with on their screen; the instance itself is not visible. Next, we will add a TextLabel instance and name it Message. This label will display the text that the server assigns to the StringValue instance named Message, which you created and added to ReplicatedStorage. After doing this, add another TextLabel instance named Remaining. This label will function similarly to the previous one but will display the text of the StringValue in ReplicatedStorage, named Remaining. After doing this, your StarterGui service in the Explorer should look like this:

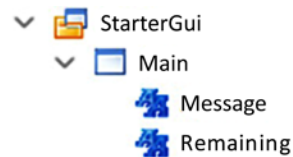


Figure 6.8: This image shows how the TextLabel instances should be parented to the ScreenGui instance

Now that you have added the necessary instances to the ScreenGui, we will need to change their properties so that they have a proper appearance on the screens of players. Let's take a look at the different properties of the TextLabel instances that should be changed from their defaults, as follows:

- The most important aspect of the TextLabel is the Text property. This property is a string, and for the purpose of this experience, it should be blank by default for both labels. Additionally, you should enable the TextScaled property of the labels, which will automatically scale the size of the font to fill the box; this property is located toward the bottom of the property list.
- The Zindex property is used to determine the order in which UI elements are displayed. For example, a TextLabel with a Zindex value of 2 would be placed above or on top of a TextLabel with a Zindex value of 1.
- The BackgroundColor3 and BackgroundTransparency properties are used to change the appearance of the background of several UI instances, using a Color3 value and a number respectively. For our uses, we will be setting the background of the Message TextLabel to be semi-transparent and gray, but you may select any color you wish.

Furthermore, since we will be working with a gray background, we will want to set the `TextColor3` property to a lighter color, preferably something closer to white so that the text is visible.

- The `Size` and `Position` properties are used to determine the appearance of UI instances. These properties take a `UDim2` userdata, which, like most userdata types, is constructed via the `.new()` constructor. A `UDim2` consists of four elements, which alternate between two size types called **scale** and **offset**. When a UI instance is sized with scale, it will have a size relative to your screen; when sized with offset, it is sized directly in pixels. For this experience, we will only use scale, meaning that if we wanted to create a value for `Size` that covers the entire screen, we would do `UDim2.new(1, 0, 1, 0)`.
- The `AnchorPoint` property is part of many UI instances. This property is a `Vector2` userdata and changes the point at which the interface is positioned, as well as how the interface expands or contracts with size changes. For a UI element to be universally centered, the `AnchorPoint` property would need to be `Vector2.new(0.5, 0.5)`, and the `Position` property would be `UDim2.new(0.5, 0, 0.5, 0)`.

Looking at *Figure 6.9* below, we can see that we will need to manipulate all of the listed properties. You do not need to set these properties in a script and are, in fact, encouraged to just set them from the Properties menu of Studio. The `AnchorPoint` property of the `Message` label should be `Vector2.new(0.5, 0.5)`. This label will span the entire width of the screen and serve as the background for both labels. To do this, we will first set the `BackgroundColor3` and `BackgroundTransparency` properties to `Color3.fromRGB(86, 86, 86)` and `0.5` respectively. Once this is complete, we will change the `Size` of the `Message` label to `UDim2.new(1, 0, 0.1, 0)` and the `Position` to `UDim2.new(0.5, 0, 0.05, 0)`; this should make it span the width of the screen and be placed at the top and center of your screen. With these changes, the `Message` label is good to go.

Now, let's look at the `Remaining` label. This label will have the same `AnchorPoint` property as the `Message` label. The `BackgroundTransparency` property, however, should be `1`, as we are using the background from the `Message` label as the background for this label. Once these changes have been made, change the `ZIndex` value of the label from `1` to `2` to ensure that this label is displayed in front of the background. Lastly, we will need to change the `Size` and `Position` properties of the label to be nicely offset from the `Message` label. You should set the `Size` and `Position` properties to `UDim2.new(0.2, 0, 0.1, 0)` and `UDim2.new(0.85, 0, 0.05, 0)` respectively. This will set the `Remaining` label at the same height as the `Message` label, with a position offset to the right end of its background.

On a minor note, you may also want to set the `BorderPixelSize` property of both labels to 0; this property creates an outline for visibility but is not considered very attractive.

The listed properties are shown in the following screenshot:

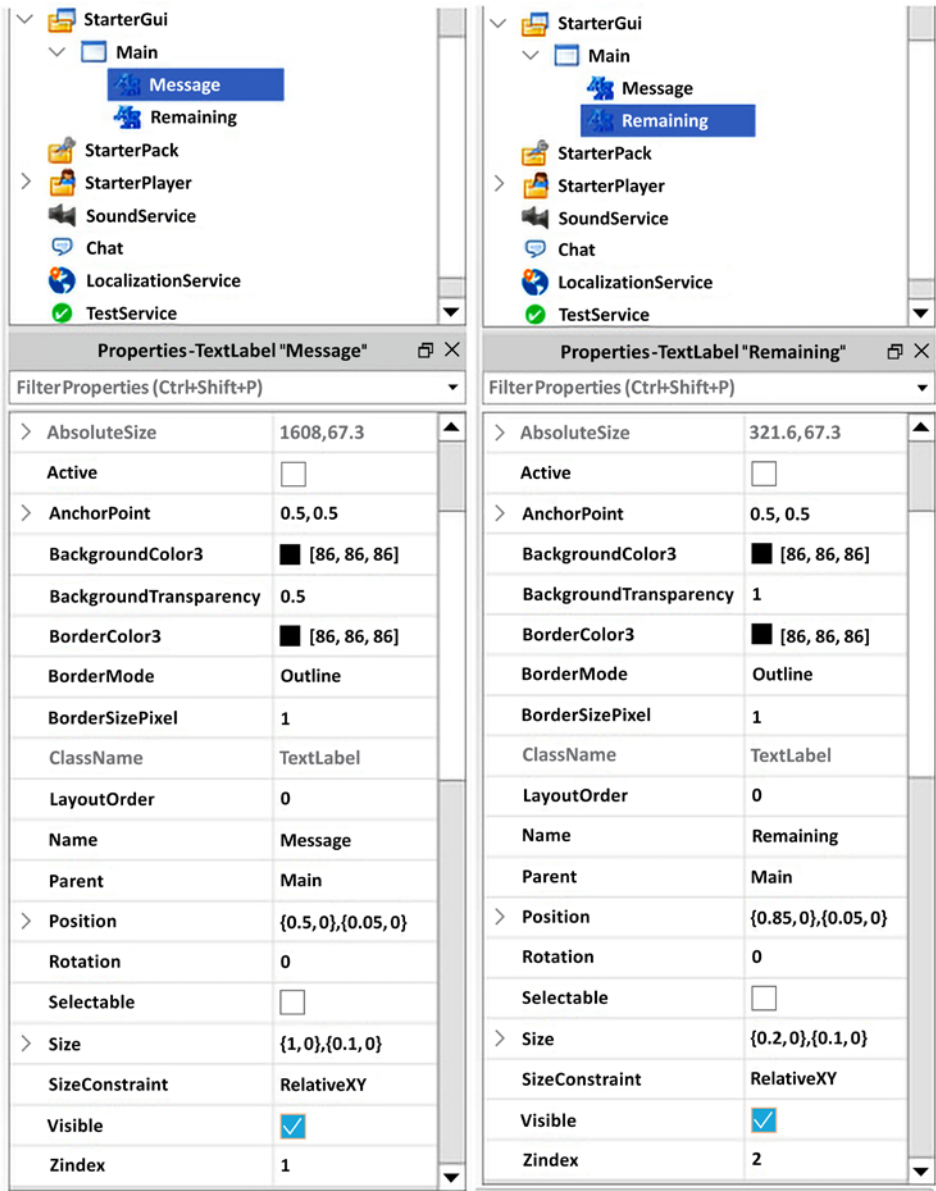


Figure 6.9: This shows the properties of the two TextLabel instances within the ScreenGui

With the UI instances now properly configured, you will need to create a new `LocalScript` and add it to the `ScreenGui` instance. This script should be titled `UIHandler`, and since we will be working with multiple UI systems, you should create it to be modular. Once again, the code that will require each module to be parented to this script has been included for your convenience here:

```
for _, module in pairs(script:GetChildren()) do,
    if module:IsA("ModuleScript") then
        task.spawn(function()
            require(module)
        end)
    end
end
```

With the `UIHandler` script now created, add a new module to it called `Display`. In the following code, you will see that we index `ReplicatedStorage`, the `ScreenGui` that the `UIHandler` is parented to, and the `StringValue` instances in `ReplicatedStorage`. For abbreviation, we also index the two `TextLabels` that will be used to display the values of the `Message` and `Remaining` `StringValue` instances, as shown in *Figure 6.10*:



Figure 6.10: The two `StringValue` instances in `ReplicatedStorage`

Next, by way of the `GetPropertyChangedSignal()` method of instances, we will update the `Text` property of the `TextLabel` instances to the new value held by the `StringValue` instances only when their `Value` property changes. We implement this event function rather than a loop because it is more performant. This way, we are only making assignments when the values change rather than over any given interval when we do not necessarily need to. You should add the following code to the module, without alteration, unless you know a specific change you want to make to the displayed text:

```
local replicatedStorage = game:GetService("ReplicatedStorage")
local gui = script.Parent.Parent
local message = replicatedStorage.Message
local remaining = replicatedStorage.Remaining
local display = {}
local messageLabel = gui:WaitForChild("Message")
```

```
local remainingLabel = gui:WaitForChild("Remaining")
messageLabel.Text = message.Value
remainingLabel.Text = remaining.Value

message:GetPropertyChangedSignal("Value"):Connect(function()
    messageLabel.Text = message.Value
end)

remaining:GetPropertyChangedSignal("Value"):Connect(function()
    remainingLabel.Text = remaining.Value
end)

return display
```

The next topic we will cover will be the UIs that could keep players in your experience even if they continuously lose.

## Making a spectate menu

As mentioned before, when competitors die and return to the lobby, you will want to keep them engaged so that they do not simply leave your experience out of boredom or anger. A good system to implement is a spectate menu, whereby players will be able to view the remaining competitors and cycle through them with the simple click of a button.

Since we will be working with multiple UI elements that we will want to make visible or invisible all at once, we will need to implement a `Frame` instance. This instance is most often used to serve as a container for other UI instances. The reason we want to implement a `Frame` instance here is that we can instantly make all of the UI elements associated with this system appear or disappear by changing the `Visible` property of the `Frame` instance; this reduces the amount of code you must write and improves physical organization, as visible in the Explorer. You should name this `Frame` instance `Spectate` and universally center it, make its size the full size of the screen, and make its background transparent. This will ensure that there is no difference in behavior for scaling or position when UI elements are parented here.

You can see the properties of a properly configured Frame instance in the following screenshot:

Properties - Frame "Spectate"	
Filter Properties (Ctrl+Shift+P)	
Active	<input type="checkbox"/>
> AnchorPoint	0.5,0.5
BackgroundColor3	<input type="checkbox"/> [255,255,255]
BackgroundTransparency	1
BorderColor3	<input checked="" type="checkbox"/> [27,42,53]
BorderMode	Outline
BorderSizePixel	0
ClassName	Frame
LayoutOrder	0
Name	Spectate
Parent	Main
> Position	{0.5,0},{0.5,0}
Rotation	0
Selectable	<input type="checkbox"/>
> Size	{1,0},{1,0}
SizeConstraint	RelativeXY
Style	Custom
Visible	<input checked="" type="checkbox"/>
Zindex	1

Figure 6.11: This screenshot shows what the properties of the Spectate Frame instance should be

After creating and sizing your Frame instance, you will need to create three TextButton instances and one TextLabel instance. TextButtons are like TextLabels but have the ability to detect when they are being interacted with via events.

Since the `TextButton` instance is part of the same family as `TextLabel`s, the properties are nearly identical. You should create a button for toggling the spectate UI, two buttons for going forward and backward through the remaining players, and one `TextLabel` instance to display the name of the player you are currently spectating. You may notice that this UI is best described as aesthetically primitive. If you want to learn how to beautify your UI, I suggest referring to the following articles and others like them on the developer website:

<https://developer.roblox.com/en-us/articles/Intro-to-GUIs>

<https://developer.roblox.com/en-us/articles/Creating-GUI-Buttons>

The following screenshot shows how I configured my UI. This is a good layout for several reasons, including it being intuitive for the user, with clearly shown buttons that indicate what they do by appearance as well as showing some information about who they are spectating. While the layout of the following UI is good, it is basic, and the design of the buttons is primitive. When making a project of your own, you should make—or hire someone to make—a UI that is colorful and engaging for your players:



*Figure 6.12: This shows the Spectate Frame instance and toggle buttons, and Message and Remaining TextLabels*

The first step in creating this new system will require that you add a new module to the `UIHandler` script, named `Spectate`. You can see in the following code that we define which services will be needed for the system as well as the `Frame` and the UI elements within it. Moreover, we create some other variables that will be used to help us track information when the system is being used. You should add the following code to the module:

```
local replicatedStorage = game:GetService("ReplicatedStorage")
local playerService = game:GetService("Players")
local player = playerService.LocalPlayer
```



```

local cam = workspace.CurrentCamera
local spectate = {}
local gui = script.Parent.Parent
local spectateFrame = gui:WaitForChild("Spectate")
local toggle = gui:WaitForChild("Toggle")
local nameLabel = spectateFrame:WaitForChild("NameLabel")
local nextPlayer = spectateFrame:WaitForChild("NextPlayer")
local lastPlayer = spectateFrame:WaitForChild("LastPlayer")
local competitors = {}
local curIndex = 1
local spectating = false
spectateFrame.Visible = false

return spectate

```

First, the system needs to be able to see which players are still alive. Luckily, we can easily figure out that the players who are still alive are those contained within the `competitors` table in the `GameRunner` module. So, to get this table to the client from the server, we will implement a new `RemoteFunction` parented to `ReplicatedStorage` and name it `GetCompetitors`. Additionally, we need to add a new `RemoteEvent` parented to `ReplicatedStorage`, named `UpdateCompetitors`.

In the following code, we have two parts to the code block—one part denoted by `client` and one part denoted by `server`. The client portion of the code should be added to the `Spectate` module, as it will set the `competitors` variable to the table returned by the `RemoteFunction` invocation and will update the table when the `RemoteEvent` is fired. The server code will go in the `GameRunner` module and return the `competitors` table back to where the invocation was made. The `getCompetitors()` function will be called when the `spectate` UI is toggled as well as whenever the player clicks to cycle through the remaining players, causing the `competitors` table to update; we will handle that logic in the upcoming examples.

For the `UpdateCompetitors` `RemoteEvent`, the server will fire this instance to update the `competitors` table on each client when the round begins and when competitors die. To do this, make sure to add the line `updateCompetitors:FireAllClients(competitors)` to the `removePlayerFromTable()` and `gameLoop()` functions of the `GameRunner` module for when a player is actually removed, right after players are added to the `competitors` table via `addPlayersToTable()` and, when the round ends, after the `competitors` table is made blank.

The following code should be added to the two modules, without alteration:

```
--client
local getCompetitors = replicatedStorage.GetCompetitors
local updateCompetitors = replicatedStorage.UpdateCompetitors
spectate.getCompetitors = function()
    competitors = getCompetitors:InvokeServer()
end
updateCompetitors.OnClientEvent:Connect(function(list)
    competitors = list
    for _, competitor in pairs(competitors) do
        if competitor == player then
            toggle.Visible = false

            if spectating then
                spectate.toggleSpectate()
            end
        end
        return
    end
end)
if spectating then
    spectate.focusCamera(competitors[curIndex])
end
end)

--server
local getCompetitors = replicatedStorage.GetCompetitors
local updateCompetitors = replicatedStorage.UpdateCompetitors
getCompetitors.OnServerInvoke = function()
    return competitors
end
```

Working back in the Spectate module, the following function, `toggleSpectate()`, will make two cases based on whether the spectate system is currently being used or not. If the system is not already in use when the function is called, then the `spectating` variable is set to true, the `competitors` table in the module is updated, the UI associated with the system is made visible, and the client's camera is focused on the first player within the `competitors` table. If the spectate system is being used when the function is called, then the `spectating` variable is set to false, the UI is hidden, and the client's camera is refocused on themselves.

The following code should be added to the Spectate module, without alteration:

```
spectate.toggleSpectate = function()
    if not spectating then
        spectating = true
        spectate.getCompetitors()
        spectateFrame.Visible = true
        local targetPlayer = competitors[1]
        spectate.focusCamera(targetPlayer)
    else
        spectating = false
        spectateFrame.Visible = false
        spectate.focusCamera(player)
    end
end
```

The `focusCamera()` function is used to focus the client's camera on the character of a passed player. This function also checks if the number of competitors remaining is 0 and toggles off the spectate menu as a round is not currently in progress. If there are players remaining, however, then the client's camera is focused on that player, and the `TextLabel` property of the `NameLabel` `TextLabel` is updated. You should add the following function to your module:

```
spectate.focusCamera = function(targetPlayer)
    if #competitors == 0 and spectating then
        spectate.toggleSpectate()
    else
        if targetPlayer then
            cam.CameraSubject = targetPlayer.Character
            nameLabel.Text = targetPlayer.Name
        else
            spectate.getCompetitors()
            local newTargetPlayer = competitors[1]
            spectate.focusCamera(newTargetPlayer)
        end
    end
end
```

The following code adds functionality to the UI buttons you have now added. By using the `MouseButton1Click` event of interactable UI instances, we can create a behavior whenever a UI button is clicked. Note that when the button assigned to the `toggle` variable is clicked, this will call the `toggleSpectate()` function of the module. For both of the buttons assigned to the `nextPlayer` and `lastPlayer` variables, we update and cycle through the `competitors` table, checking if the `curIndex` variable needs to be reset if it is incremented to a non-existent table position. If the index is valid, then the player's camera is focused on the player at the given index in the `competitors` table. The code for this is shown here:

```
toggle.MouseButton1Click:Connect(function()
    spectate.toggleSpectate()
end)
nextPlayer.MouseButton1Click:Connect(function()
    spectate.getCompetitors()
    curIndex += 1
    if curIndex > #competitors then
        curIndex = 1
    end

    local targetPlayer = competitors[curIndex]
    spectate.focusCamera(targetPlayer)
end)

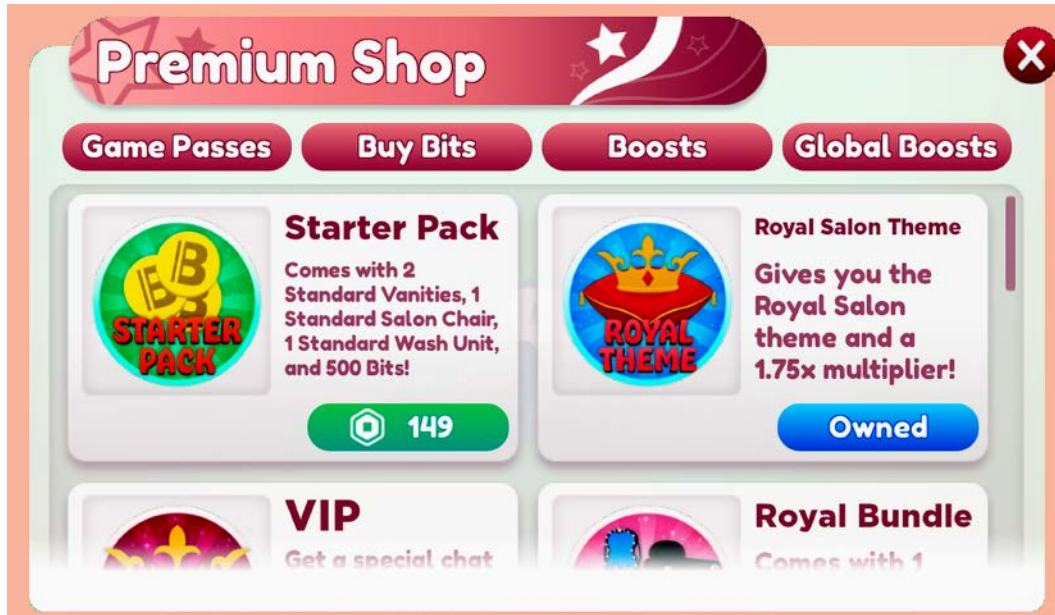
lastPlayer.MouseButton1Click:Connect(function()
    spectate.getCompetitors()
    curIndex -= 1
    if curIndex < 1 then
        curIndex = #competitors
    end

    local targetPlayer = competitors[curIndex]
    spectate.focusCamera(targetPlayer)
end)
```

## Creating a shop

If you want to create a shop for your experience, you can now create a UI for it or follow the method using parts, as you were taught in *Chapter 5, Creating an Obby*, in the *Shops and purchases* section.

It should be noted, however, that in most experience formats, users are more likely to notice an attractive button that is readily on their screen compared to a prompting Part, which they may not find during their play session. For this reason, I encourage you to make your own UI-based shops for purchases made with both Robux and in-experience currency. You can see an example of a UI-based shop in *Figure 6.13*:



*Figure 6.13: This shows a UI-based in-game shop*

Remember that to manage purchases made with Robux, you can simply copy and modify the Monetization module made in the previous chapter.

## Summary

In this chapter, you learned how to manage players in a round-based game, created weapons with client security, learned about the concept of local replication, and made a functionally engaging UI. You now have the skills to make any experience you can think of on the diverse Roblox platform. Before you begin making projects of your own, be sure to review the key concepts from these experience example chapters, including security, engaging interactions, and maintaining a clean and organized code base.

In the next chapter of this book, you will learn the best practices for popularizing and sustaining the experiences that you will make in the future. In particular, you will focus on how to promote and grow your experiences, monetize them as optimally as possible, and implement features that retain and excite your current and potential audiences.

## Invitation to join us on Discord!

Book club is a great platform to learn from other users and share your games for feedback and invite others to play along.

Join us on the community platform to ask questions, share your experience building games on Roblox thus far, and participate in *Ask Me Anything* sessions and more events with the author.

Scan the QR code or visit the link to join the community.



<https://packt.link/letscreategames>

## Worksheet

- Q1. Why is it good to keep weapon settings in a module?
- Q2. Why is a lobby necessary in experiences of this format?
- Q3. What do you need to have in your map for every player that can be in your experience at once?
- Q4. Why might it be good to include a celebration message or effect showing the victor at the end of a round?
- Q5. Why is local replication important for different effects?
- Q6. Where can you find many quality weapon models in addition to millions of other assets?
- Q7. What can you add to rare weapons and items to make finding them more exciting?
- Q8. Why might a UI-based shop be better than a physical shop?



---

# Section 3

---

## The Logistics of Game Production

This section will focus on the logistics and business skills needed to ensure that a project progresses from start to finish. It will cover the topics of budgeting, time management, development efficiency, and collaboration, which are particularly critical in the Roblox development sphere. Additionally, we will explore different ideas to inspire you when making experiences of your own.

This section comprises the following chapter:

- *Chapter 7, The Three Ms*
- *Chapter 8, 50 Cool Things to Do on Roblox*





# 7

## The Three Ms

Through the previous chapters, you have learned how to program in Luau and create engaging experiences on Roblox. With the skills you have, you can create an experience around virtually any vision that comes to mind. However, it is one thing to create an experience of your own, but to find success with it you need to follow a set of steps to optimize your experience's performance.

The main performance categories for an experience typically rely on three alliterative factors, those being **Mechanics**, **Monetization**, and **Marketing**—or, as I have coined them, the **Three Ms**. This chapter will focus on introducing you to the best practices to follow regarding these three groups, to ensure that your experience is as enjoyable as possible for players, your earnings are maximized without intruding on the user experience, and your experience captures and retains the largest audience possible.

In this chapter, we're going to cover the following main topics:

- Mechanics
- Monetization
- Marketing
- Reviewing what you've learned

### Technical requirements

This chapter will focus entirely on the best practices for finding success on the Roblox platform. Some articles from the developer website and the **Developer Forum (Dev Forum)** will be linked throughout the chapter, so having an internet connection to view these or to research any other mentioned topic would be beneficial.

## Mechanics

The first *M* we will cover is mechanics. The mechanics of experiences on Roblox, or any platform for that matter, can vary widely. Some popular themes on Roblox include **simulators**, **role-play (RP) games**, **tycoons**, and **minigames**. The shared characteristics between these experiences regarding mechanics can be simplified down to repeated reward, social interaction, or both.

In this section, we will break down the mechanics of these commonly occurring experience themes and identify what makes them successful so that you can potentially integrate those ideas into entire genres of your own creation.

## Simulators

For experiences such as simulators, users start off with very basic tools, weapons, or abilities. Over time, as users perform some simple task—often as simple as clicking their mouse—they gain stats, which can be converted to a currency that is used to purchase better items in the experience. This allows players at the beginning of the experience to make progress more quickly. Of course, the ability to make progress *more quickly* is something of an illusion, as the price of items increases exponentially, forcing you to *grind* (perform repetitive tasks to achieve an outcome). The genius of this format is that it allows new players to progress rather rapidly while they are consistently being rewarded with what seems like great amounts of currency. As the requirements for reaching the next progress marker increase, players seek the easily attained success they were conditioned to when they began playing the experience, encouraging them to continue playing.

Have a look at the following screenshot:



*Figure 7.1: RussoPlays completes a beginning quest while showcasing Power Simulator in a video*

Despite capturing and often firmly holding an audience, simulators do not have an infinite lifespan. Without consistent updates, audiences will become bored, especially if they have acquired all unlockable content within the experience. The trick to keeping a simulator alive is to provide new content frequently at the beginning and then slowly less as you transition to other projects. With each update, you should add new unlockable items and other free content, as well as some new monetization options. A famous example of a long-lived experience of this genre is *Bubble Gum Simulator*, by *Rumble Studios*. It was released in 2018 and still receives tens of thousands of concurrent players every day; they have managed to do this by following the preceding steps.

## RP games

RP games have been on the Roblox platform for some time, but recently they have begun to dominate the **Popular** sort, with new renditions of the concept. RP games should not be confused with **role-playing games (RPGs)**, which are typically quest-based games, which involve unlocking different items and discovering many places as you explore. RP games are experiences where users simply fulfill some role, typically in a peaceful setting. Some famous examples are experiences that take place in cities or neighborhoods and that are focused around doing fun tasks or becoming a member of a family, such as *Adopt Me*, *Meep City*, *RoCitizens*, and more.

These experiences are particularly successful because players can do many non-repetitive things and the experiences have no end. While there are often things to discover and explore, most of the experience is oriented around interactions with other people and improving personal items, such as raising pets or customizing your house. These experiences, as with simulators, still need to be updated and, similarly, updates should include new free content as well as new monetization choices.

## Tycoons

In recent years, tycoons in the traditional sense have become less popular on the Roblox platform, but they still exist in a slightly different format compared to their original counterparts. Tycoons are experiences where players add different items—typically machines—to a home base in order to earn more currency. With more currency, players can, in turn, buy more decorative and functional objects for their base, expanding until they dominate the server. While not all tycoons revolve around combat, there is almost always some element that keeps players competing against each other outside of showing who the richest player on the server is. The following screenshot gives an example of a typical tycoon experience:



Figure 7.2: An example of an experience of the traditional dropper tycoon genre, *Miner's Haven*

These experiences have fallen somewhat out of favor because once a tycoon is complete and all items are unlocked, there is not much of an incentive to continue playing. Updating these experiences can be a more difficult task, depending on how your tycoon is structured. When updating, it is best to add new items to your tycoon, new map features, and new additions to monetization. Inevitably, these experiences tend to lose their peak success after a comparatively shorter length of time than experiences of the other genres that have been mentioned.

In modern renditions of tycoon games, developers will often take advantage of **non-player characters (NPCs)**, more complex ways of unlocking new items, infinite expansion capabilities, and sometimes stunning visuals. A good example of this *modern* type of tycoon would be *My Restaurant*, by *BIG Games*.

## Minigames

Minigames or any sort of round-based, arcade-styled experience have been continuously popular on the platform but have not seen the same magnitude of success as the aforementioned genres. These arcade-like experiences are easier to update as you can simply add new levels or maps. Additionally, as with RP games, these experiences are incredibly social, with many players making sure to specifically join their friends or making use of **parties** (groups of players).

If you wish to make a round-based game, ensure that the social aspect is the focus, with many monetization features that do not interfere with the user experience. To do this, you will learn how to adopt good monetization strategies for this genre and many others in the following section.

## Monetization

The next *M* is monetization. Monetization is one of the most important aspects of development if your primary goal is to make Roblox either a career or a profitable hobby. The first thing to realize is that there are great and terrible implementations of monetization. For example, all players, regardless of whether they buy anything in your experience or not, can be positively or negatively impacted by your monetization strategy. We will start off with what *not* to do.

Perhaps the most egregious offender that you will see players mention is “pay-to-win” monetization. In these instances, players that have the ability to spend money can purchase very powerful items, or anything that gives them a unique advantage over free players. Some game companies such as **Electronic Arts Inc. (EA)** are notorious for this style of monetization and it has, in the past, stunted the initial—and later, overall—performance of some of their games, such as *Star Wars: Battlefront II*, which had an overbearing *loot crate* system. When monetization is implemented like this, it often leads to ruining the experience for all but the players who purchased the premium items and, even then, it is no more enjoyable for them to play against others who have bought the same perks.

Making Passes that give an advantage is still acceptable, but the abilities of these items should be somewhere in the mid- to upper-mid-tier range. This is an especially important practice to observe if your experience has any match-making systems based on skill, as new players can otherwise buy their way to the top instantly.

After fallout with their player bases due to overbearing monetization strategies, game development companies big and small have responded by shifting their main purchasable items to content expansion or cosmetic options.

Cosmetic options are perfect for avoiding the previously mentioned problems, as choosing to make the purchase is purely based on personal preference and no advantage is gained or lost by purchasing or not purchasing the item.

For content expansion, this approach can also be implemented well, especially on Roblox. Normally, content expansion is accepted throughout various player communities but not to the extent that you must essentially purchase the game twice to unlock a full game experience. This error has also been made by triple-A game studios in the past, with detrimental effects on the game's performance. However, Roblox experiences are almost always free, outside of the few that choose to use the paid access system, which was mentioned back in *Chapter 2, Knowing Your Work Environment*. Because Roblox is oriented around free-to-play experiences, having some content behind a paywall is even more acceptable than it would be elsewhere. A good application of this occurs in some simulators, where players can access an area that gives them slightly more resources and cooler aesthetics only if they own a Pass; typically, something such as this would come with a VIP Pass. In review, both are viable strategies, and you should implement them as you see fit and monitor the response from your community.

On the topic of a monetization option such as VIP, sometimes it can be more advantageous to structure some paid benefits as a subscription as opposed to a single-purchase Pass. For example, let's say a VIP pass grants players in your experience a special chat tag, a 1.5x boost for whatever stat, and access to a special location. Something such as this could be priced around 400 Robux for some experiences. However, data has shown that turning these benefits into a subscription, whereby players can purchase a product to add on to their remaining VIP time, can actually increase profits by a considerable margin over a similar time interval as compared to selling a Pass version. Currently, to make a subscription system, you need to record the times at which products are purchased and save that information to the player's data. This is not a very complicated process, but nor is it the most convenient. Because of this, Roblox has listened to its developers and plans to devise a new subscription system and implement it into its **application programming interface (API)** so that developers do not need to do any of the backend themselves. At the time of writing, this system has not been released to developers in any form, but you can look up some potential future uses of it by visiting the developer website using the following link and searching for **subscription** within the page: <https://developer.roblox.com/en-us/api-reference/class/MarketplaceService>.



## Marketing

Finally, the last *M* is marketing. Marketing is the process through which you will make your experiences known to the world and gain thousands of new players. We have touched on some marketing concepts before, but let's look at what makes a marketing campaign successful and how to engage the most players at a minimal cost.

### The Roblox promotion system

As previously discussed, Roblox has an internal system of promotion that utilizes two forms in which promotional material appears to players; these are called **User Ads** and **Sponsors**. In *Figure 7.3*, you can see an example of an experience promoted via the **Sponsors** system and an advertisement made for an experience using the **User Ads** system:

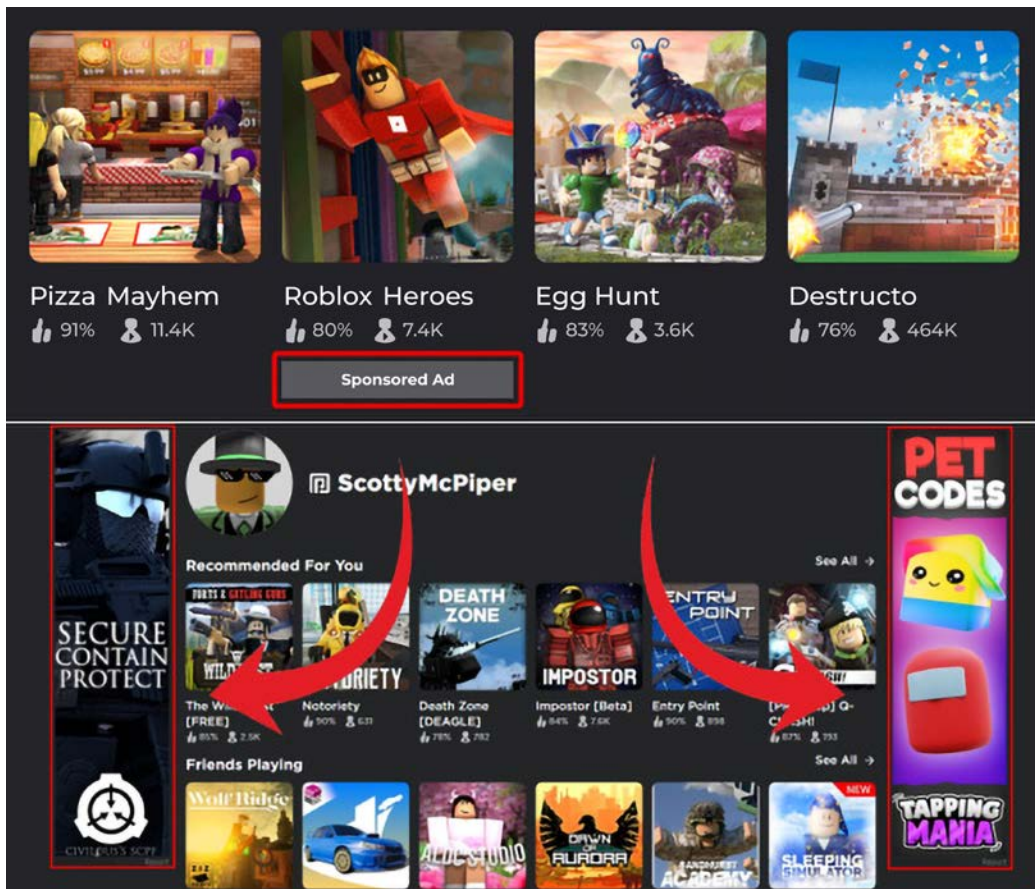


Figure 7.3: This shows examples of the **Sponsors** (top) and **User Ads** (bottom) promotion systems

There are advantages and disadvantages to using one of these systems over the other. When using the Sponsors system, advertisement-blocking web browser extensions will not stop the promotional material from showing, as they do with User Ads. This can have a big effect because, as of recently, the User Ads system stopped displaying contents on the **Games** page, and now ads can only be found in certain locations on the website.

In the case of User Ads, you can create a custom image, matching specific dimensions, to advertise your experience. This can be especially beneficial if your experience is not defined by a single image, such as the icon of your experience, which is what is displayed when using the Sponsors system. Often, developers will make anything from renders, to memes, to full comic strips using this system, to give their experience more character at a first glance compared to that achieved by an experience icon using the Sponsors system.

Toward the end of 2020, however, a change to the **Games** page was made that has severely affected the amount of impressions developers receive for the Robux they spend. Like the User Ads system having its display locations limited, a change in the order of sorts on the **Games** page has made it so that players will only see sponsored experiences if they scroll down to the **Popular** or **Most Engaging** sort, and any others further down. This has had some serious and immediate negative effects, with some developers seeing significant engagement with their promotional material lost since the change. This is unfortunately removing the traditional avenue through which developers made their experiences popular.

Roblox has, however, made some positive changes to improve the current state of discoverability. In 2020, they rolled out the current system that allows for promotional campaigns to be targeted at specific, but limited, demographics so that you can best capture your potential audience and your efforts are not spent on someone who is less likely to be interested in the content you create. More recently, they've added many more sorts to the **Discover** page, meaning that your experience has more places to be potentially featured. As a final piece of advice for this section, you should ask yourself where you want to distribute your marketing money in order to engage the most players. This really depends on how compatible your experience is with different devices. In most cases, for your experience to be the most enjoyable on platforms other than PC, in other words, mobile, tablet, and Xbox, you may need to make special cases for controls as well as more peripheral features, such as changing the appearance of the **user interface (UI)** on screens. With the new features just mentioned, you should also decide which demographic is most likely to enjoy your experiences, whether that is based on age, gender, or anything else.

Moving forward, we will discuss an alternative form of promotion to reach much further than you would traditionally be able to with the Roblox promotion system, all the while avoiding the current setbacks it has.

## YouTubers

Because of the previously mentioned factors and some older trends on the Roblox platform, YouTubers are becoming a major component of any marketing campaign. As mentioned in *Chapter 1, Introducing Roblox Development*, the relationship between Roblox developers and YouTubers is symbiotic; as developers make experiences, it gives YouTubers content to film, and in turn, subscribers to that YouTuber often flock to the experience that was featured.

Outside of general exposure, sometimes YouTubers are in search of developers to create experiences for their audiences to play or so that they can host events. When opportunities such as these occur, you should take them if a fair deal is offered. Not only do you promote yourself and your development abilities, but you also make strong connections with those influencers, who will likely be willing to make videos on experiences you make in the future. An example of this was my own partnership with *Tofuu*, whose channel is seen in *Figure 7.4*. Tofuu was looking to create a experience with a team of developers, and together we created *Munching Masters*, one of my most popular experiences to date, which has had more than 120 million visits.

It should be clear that you do not need to make a full project with influencers to establish a connection with them. The biggest professional tip I can give in this book is to simply reach out. You will be surprised how often simply taking the initiative of reaching out to someone will return a positive response. Remember, the worst that can happen in any of these cases is that your offer is simply declined.

Tofuu's channel is shown here:

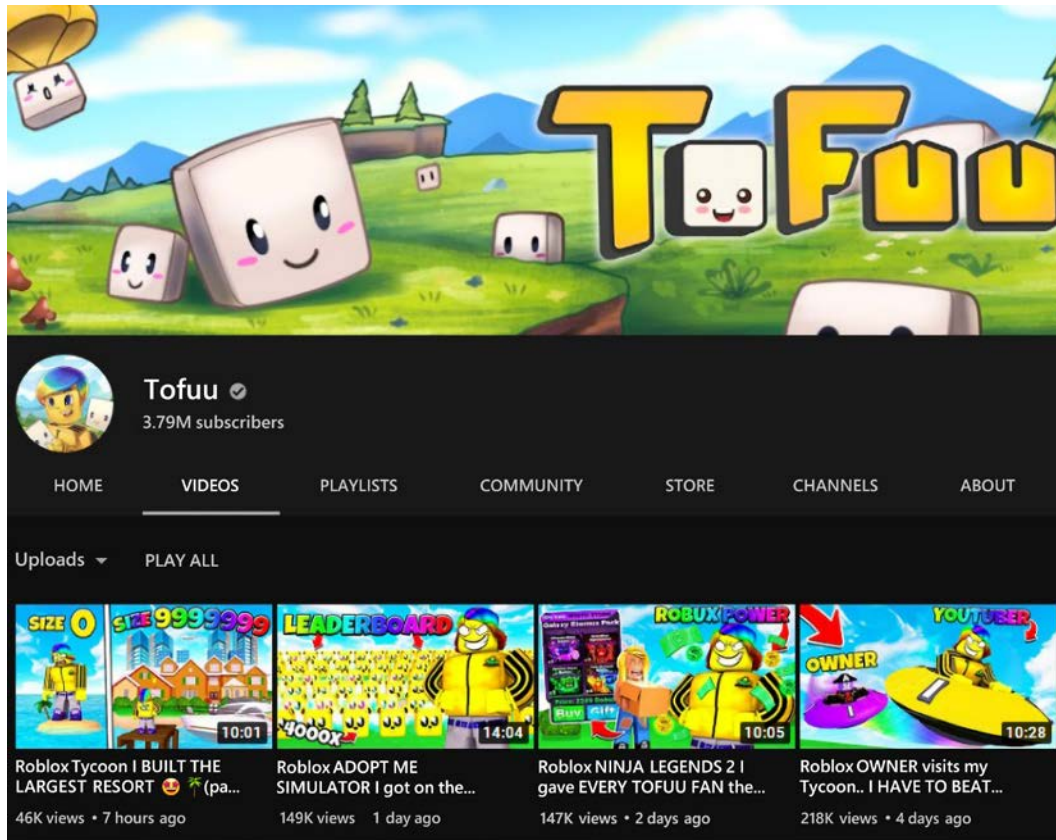


Figure 7.4: Tofuu is a Roblox YouTuber who often makes videos on experiences with unique mechanics

## Community

Creating a community around your work can be very important to your experience's success. As mentioned in the first chapter, creating a Discord server for your experience or simply engaging with members of your Roblox group can help your experience maintain a more consistent player base. Once you develop your community, you can begin building excitement or “hype” for your new releases and be guaranteed additional players at the launch of your experience.

One way to build this hype is by posting sneak peeks and teasers for your new experiences or updates of already released titles. These can be any sort of media, like videos or images, but the less you reveal, the more curious your audience may be and the more anticipation grows. You should post these teasers wherever possible, especially taking advantage of social media. If your post gets traction and you use hashtags appropriately, you may also attract new Roblox players to your experience.

Let's now take a look at everything we've learned so far.

## **Reviewing what you've learned**

With the conclusion of this chapter, you have completed this course in Roblox development. You have gone from having what is likely little to no experience working with Roblox to possessing the tools necessary to create your own career. Let's look at some of the skills you gained chronologically, as follows:

### **Chapter 1**

In the first chapter, you learned what the Roblox platform is about and what it has enabled others to accomplish, and what you can accomplish, through learning and practice. Additionally, we discussed the different types of developers that exist on the platform and the skillsets that each of them grants you, should you choose to take them up. Lastly, we talked about the professional skills to be gained that you can take into the future with you, regardless of your career trajectory. Overall, this chapter informed you why there is so much to be earned from the platform and what it has already done for others.

### **Chapter 2**

In the second chapter, you learned about Roblox Studio, the beginning of the inner workings of Roblox development, and what it means to be a Roblox developer. Covering material about the Roblox website, the tools developers use to create their experiences' physical structures, as well as establishing general experience and place settings, this chapter was critical for setting you on your path to succeed in the following chapters and your career as a developer.

### **Chapter 3**

The third chapter was the most crucial part of this book for your future as a developer. This chapter covered numerous topics while introducing you to the Luau language; not only will learning Luau be key for creating experiences on Roblox, but it will also provide an excellent base for any programming you do in the future.

Moreover, you learned many universal constructs in programming and computing, including programming knowledge that can be applied to many languages, how your computer processes different datatypes, the client-server relationship, and much more. After introducing various datatypes, you learned how to manipulate them; in the case of CFrames and vectors, you developed the skills to work in 3D environments, an invaluable ability to learn early on should you enter a STEM (science, technology, engineering, and mathematics) field.

## Chapter 4

The fourth chapter taught you more outside of general programming. To be specific, you learned how to manipulate the physical aspects of your experience's environments, how to improve and add effects to different systems, and how developers create many experience behaviors by way of Roblox services and APIs.

## Chapters 5 and 6

Chapters 5 and 6 focused on creating experiences, using all of the knowledge you had learned up until then. These chapters expanded your skills by demonstrating how to accomplish behaviors that are common throughout many experiences, so that you are best enabled to make whichever kind of experience you envision in the future. For example, the way that the Obby and Battle Royale games are dynamically structured allows code, such as that contained within the Datastore or Monetization modules, to simply be added to future projects, with little to no time spent modifying them. Whenever you are in doubt, you will always be able to refer to these chapters to observe the skills in programming style, use of services, and overall experience organization.

## Metaverse

“Metaverse” can be considered an honorary fourth *M*. While this topic isn't core to your success as a developer like the other *Ms*, being familiar with it might give you some perspective on the future of Roblox and what types of experiences you want to create.

The metaverse, as a concept, is a network of virtual experiences focused on social connection and interaction. While the idea of the metaverse has been around for decades, interest in practical versions of it began to grow exponentially in 2021 and onward. While most think of the metaverse as a recreational space, some companies or individuals, like Mark Zuckerberg and Meta, started work on building environments for nearly every aspect of everyday life to be lived virtually. With this spike in consumer interest and potential reimagining of the human experience, investors and companies alike began putting large amounts of money into developing this concept.

While many platforms have emerged, focusing primarily on being built with blockchain technology, the prospect of Roblox becoming a leader in this space has also become a popular idea.

While many of the competing platforms are new, Roblox has long been established as a popular social platform with an existing infrastructure, capable internal teams, and a great deal of funding. When Roblox joined the New York Stock Exchange and became a publicly traded company in 2021, there was an influx of top game studios as well as brands making an entrance to capture new audiences in the metaverse. Roblox, in turn, has put special focus on supporting events for different brands and entertainers as well as new social features, such as voice chat, to cultivate more social interaction.

What you should take away from this is what new jobs and creative opportunities are becoming available, how Roblox will continue to grow in the future, and how metaverse trends may affect the world of gaming and the way people live their everyday lives in the near future. Designing your experiences to have the best social interaction possible is not only good for engagement but brings us closer to some of the metaverse concepts proposed.

## Summary

In this chapter, you learned the best practices to ensure that whatever concept or theme you create in the future, it has the best chance of finding an audience and becoming profitable and popular among players from around the world. By heeding the advice of the *Three Ms* (Mechanics, Monetization, and Marketing), I have no doubt that you will have experiences garnering thousands of concurrent users in no time. Remember that while we focused on the prospects of making money from what is otherwise a hobby, it's important to have fun and experiment with what you want to do. Make your experiences how you want them to be because the wonderful thing about topics in development theory, such as those discussed in this chapter, is that they are constantly evolving. So, do not be afraid to get out there and design a genre of your own. Make concepts and visuals that no one has ever conceived before. Be the best developer you can be from your own perspective.

In the next and final chapter, we will explore 50 cool ideas to inspire you when developing your first Roblox experiences.

## Worksheet

- Q1. What are the Three Ms?
- Q2. What is the basic game concept of a simulator?
- Q3. What is one of the greatest aspects of RP games in terms of engagement?
- Q4. What is something you should always avoid when implementing your monetization strategy?
- Q5. What is a good level of advantage to be expected from Passes?
- Q6. What is a truly fair monetization option that may bring in less overall revenue?
- Q7. In what two ways can you monetize access to your experience?
- Q8. What two types of experience promotion formats does Roblox offer?
- Q9. What is a grouping of experiences by genre, engagement, or other factors called on Roblox?
- Q10. What external group of content creators can be crucial in getting your experience publicized?
- Q11. What is one method to maximize potential performance of future experiences?
- Q12. What trend does Roblox have the potential to lead and change the way humans interact?





# 8

## 50 Cool Things to Do on Roblox

Now that you have the skills needed you create your own experiences, we'll go over 50 cool things you can do on the Roblox platform for inspiration. These topics will range from programming challenges and experience ideas to the use of new Roblox features and more.

In this chapter, we're going to cover the following main topics:

- Programming challenges
- Experience ideas
- Experience systems
- Roblox features
- Other types of development

### Technical requirements

You may want to experiment in Roblox Studio in this chapter as well as do further independent research on the topics covered. As before, you will need an internet connection to do either.

### Programming challenges

As you learned throughout this book, Luau is a fast and powerful language. In this section, we will explore more general programming challenges to increase your understanding of and proficiency in the language. Note that each example will reinforce important programming concepts for making systems in your experiences. These challenges will be roughly sorted so that easier ones are at the beginning of the section and more difficult ones at the end.

## Is number $x$ divisible by $y$ ?

We'll start simple. In a script in Roblox Studio, define a function that returns whether a passed-in number  $x$  is divisible by a passed-in number  $y$ . We define divisibility as  $x$  going evenly into  $y$ . Remember to look for possible issues with your implementation; if the user passes in 0 for  $y$ , will your function break? If yes, what cases might you consider to always return a sensible value?

## FizzBuzz

This is a programming challenge that many companies might decide to quiz you with in an interview for a programming position. The aim of the challenge is simple: write a program that prints the numbers from 1 to 100. But for multiples of three, print *Fizz* instead of the number, and for multiples of five, print *Buzz*. For numbers that are multiples of both three and five, print *FizzBuzz*. When approaching this problem, think of what mathematical operators you have and the most efficient way of checking if a number divides another.

## Find the maximum value in a table

Another simple program is finding the maximum value in a table of values. For this problem, you should make a function that accepts a table of values as an argument and returns the maximum value within it. For example, if the table passed in was {1, 5, 9, 7, 2, 4}, the program should return 9. Once you do this, make a function that returns the minimum value in the table. Additionally, try returning the index at which this maximum or minimum value occurs.

## Check whether an element exists in a table

For this problem, create a function that accepts a table and some values and returns true or false based on whether that element exists within the table. For example, if the table passed in was {1, 5, 9, 7, 2, 4} and our value was 3, the program should return false. While you can implement this function yourself, look at what functions might be available from the table library to assist you. Remember that in most instances, library functions will be more optimized than what you can implement yourself. Is there a library function that can do this task for you?

## Format seconds into hours:minutes:seconds

This problem requires you to implement a function that takes a number of seconds and returns its string representation in hours, minutes, and seconds. For example, if 30 were passed to the function, it should return "00:00:30". If 60 seconds were passed in then the function should return "00:01:00".

Lastly, if 3601 seconds were passed in, the function should return "01:00:01". You will need to consider a few cases when formatting your time where an additional 0 is necessary following a colon (:).

## Return unique elements from a table

You should now implement a function that receives a table and returns a new table containing only unique elements, that is, the set of values in the table. For example, if the table passed in is {2, 5, 9, 7, 2, 5}, the program should return {2, 5, 9, 7}. The unique elements of your output table should be ordered by the first occurrence of their values in the original table.

## Number of stickers

For this challenge, write a function that, given a Rubik's cube of side length  $n$ , finds how many colored stickers are required to cover all exposed external faces. For example, a Rubik's cube of side length 1 would need 6 stickers; of side length 2, 24 stickers; and side length 3, 54 stickers. Think about the geometry of a cube and that 1 sticker is 1 face to solve this problem.

## Concatenate two tables

This function should receive two tables and concatenate them—that is, combine them into one table. If the tables {"Hello", "Ham", "Tree"} and {10, 1, 7} were passed to your function, the output should be {"Hello", "Ham", "Tree", 10, 1, 7}. There are several valid approaches for this problem. Once you have a working solution, try making the function variadic, combining any number of tables in the order in which they were passed in.

## Reverse a table

This function should receive a table and reverse the order of the elements. If the table {26, 30, 17, 6} was passed to your function, it should return a table containing {6, 17, 30, 26}. There are again multiple valid ways to approach this problem but try to make your solution as efficient as possible.

## Sort a table using table.sort()

The `sort()` function of the table library is a convenient way to sort tables regardless of what types of data they contain. `table.sort()` takes a table and a sorting function you define; two elements are passed at a time to that function. If `true` is returned by your function, then the first element passed in is sorted lower and the second element sorted higher; the opposite occurs when `false` is returned.

To give you a more concrete example, let's say we just want to sort a table of integers to be in ascending order, we would write the following:

```
local tab = {3, 1, 2}
table.sort(tab, function(a, b)
    return a < b
end)

print(tab) -> {1, 2, 3}
```

If you do not pass a function to compare elements, Roblox will use the function we've just written by default. Try writing your own compare function that sorts a table of instances by the value of one of their shared properties.

## Sort a table using a sorting algorithm of your choice

While `table.sort()` is convenient and will work in most game development applications, there are a variety of other sorting algorithms you can implement yourself that have different tradeoffs. Some of these include quicksort, insertion sort, bubble sort, selection sort, merge sort, and more. Depending on your application, some of these algorithms may perform their sorts faster or have behaviors you might find beneficial, such as being in place or stable. If you are interested in pursuing a career in computer science or related fields, writing a sorting algorithm is a common interview question.

## Solve a linear equation

You should define a function where, given values for  $y$ ,  $m$ , and  $b$  in  $y = mx + b$ , you find  $x$ . This should be a straightforward problem but will require a little bit of algebra, specifically solving for  $x$  given the other values in the equation. This challenge will give you more practice at thinking about problems, especially when more variables are involved.

## Guessing game

This challenge takes a slight step up as you will need to receive a response from a user via a `TextBox` instance. The goal of the guessing game is to guess the number the program is thinking of in the least number of guesses possible. If the number guessed is higher than the actual number, the program should tell the user "I'm thinking of a smaller number", similarly, if the number guessed is lower than the actual number, the program should tell the user "I'm thinking of a larger number".

For this challenge, you should make a looping program that thinks of a number and continues to accept guesses and gives hints until the player guesses correctly. Separately, think to yourself what the most efficient way is to guess the actual number chosen by the program.

## Find the $n$ th number of the Fibonacci sequence

The Fibonacci sequence is a sequence of numbers where each number is the sum of the previous two. For example, the first 10 numbers of the Fibonacci sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34. You should create a function that receives a number  $n$  that represents the value of the Fibonacci sequence we want to find. For example, if we passed 7 to our function and we are treating the sequence as zero-indexed, then we would get 13. There are several ways of approaching this problem, so think about how you might implement an iterative solution or a recursive solution.

You have now sharpened your programming abilities with some real problem solving. In the following section, we will look at various ideas for systems you can create to add to your experiences.

## Experience systems

Now that we have explored different challenges in the Luau language, we will look at different systems you can create to add and enhance your experiences. These systems may be part of many experience genres, improve player experience, or improve overall engagement.

### Make a leaderboard system

Roblox is a very social platform; one of the driving factors behind the success of many experiences is competition. Adding leaderboards to your experience is a way to drive players to compete with one another, adding another element of fun while keeping players even more engaged. Your leaderboards should be put in a central location and show the top players for whichever stats you think best.

For example, in *Figure 8.1*, there are three leaderboards at the spawn area of the map keeping track of top lifetime yen, top kills, and top lifetime strength.



*Figure 8.1: Well-placed leaderboards in a simulator game*

In order to make a leaderboard system, you will need to use `DataService` as well as `OrderedDataStores` and their `GetSortedAsync()` method. You should be able to make this system using what we learned about Data Stores in *Chapter 5, Creating an Obby*, but you can learn more about `OrderedDataStores` on the page linked here:

<https://developer.roblox.com/en-us/api-reference/class/OrderedDataStore>

## Make an announcement system

An announcement system is a way for the server to show messages to players via the game chat. Adding an announcement system to your experience is an excellent way to make players more aware of events, feel more rewarded, and further fuel competition. For example, if a boss spawns in a specific location, you can have the server announce the name of the boss and where it has spawned in the chat. At the end of a race or some other competition between players, the server can announce the winner, making the winner feel more accomplished and creating more competition between players hoping to see their name announced in the chat.

You can see an example of a server announcement in *Figure 8.2*:



*Figure 8.2: The server announces that a boss a spawned in the game*

To make this system, you can use the `ChatMakeSystemMessage` option of the `SetCore()` method of the `StarterGui` service. You can learn more about it here:

<https://developer.roblox.com/en-us/api-reference/function/StarterGui/SetCore/>

## Make a daily reward system

To further increase engagement with your experience, you can add a daily reward system. With this system, players who join your experience daily will get a reward. The longer the login streak they have, the more aware of the rewards they will be, which encourages them to keep playing. Additionally, you can require players to be in your group to collect the daily reward in order to get more group members. There are many ways to approach this system, but the core elements are that the players should get a reward once daily, the rewards get better the longer their login streak is, and that they lose their streak if they don't join your experience for a day.

## Create an interaction system

Almost every experience has something players must interact with via input. For example, players might interact with a door, a lever, a pile of money, a tool, or more. You should create a system where players can interact with a physical item via a `ProximityPrompt`. `ProximityPrompts` are a type of instance that allow players to interact with different objects in their vicinity. For example, if there is a money bag on the ground, a player could approach it, be shown a `ProximityPrompt`, and be able to collect it. This system has a lot of applications as players can collect rewards, interact with input items such as buttons, levers, and switches, solve puzzles, and more.

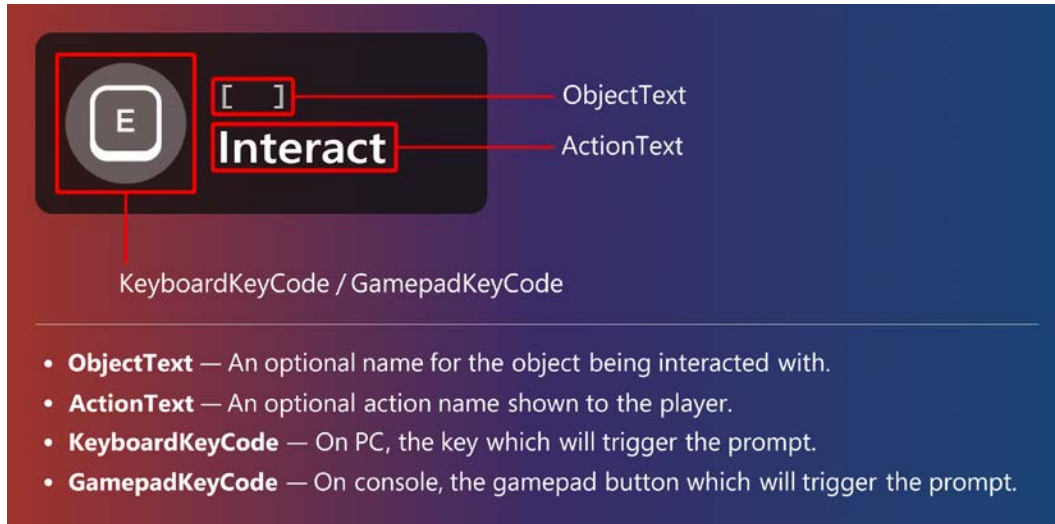
To learn more about `ProximityPrompts` and how to implement them, you can read the following article:

<https://developer.roblox.com/en-us/articles/proximity-prompts>



## Make a custom ProximityPrompt appearance

By default, ProximityPrompts have a simple UI that changes based on the properties of the prompt; you can see this default appearance in *Figure 8.3*:



*Figure 8.3: The default appearance of a ProximityPrompt*

If you want to add more style to your experience or make a more advanced system around a ProximityPrompt, you can override this default appearance with a design of your own. You can see an example of overridden ProximityPrompt appearance in *Figure 8.4* where a whole menu is loaded in addition to the presence of a button:

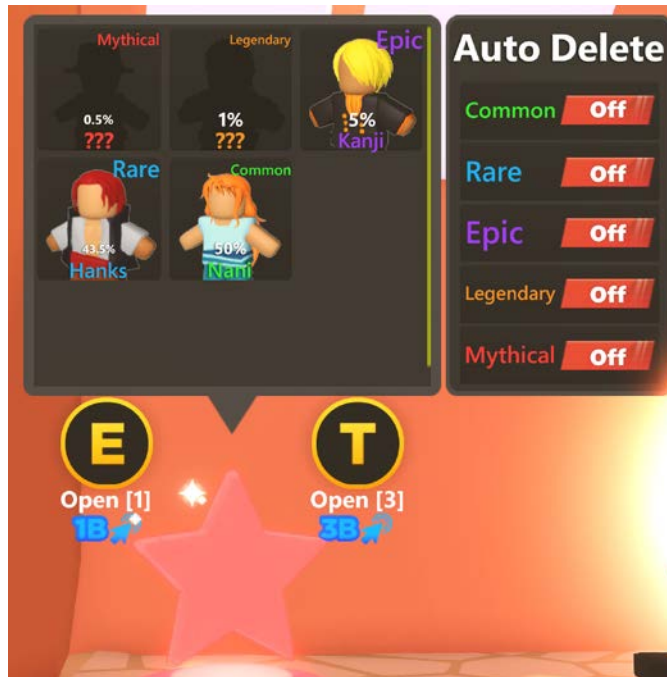


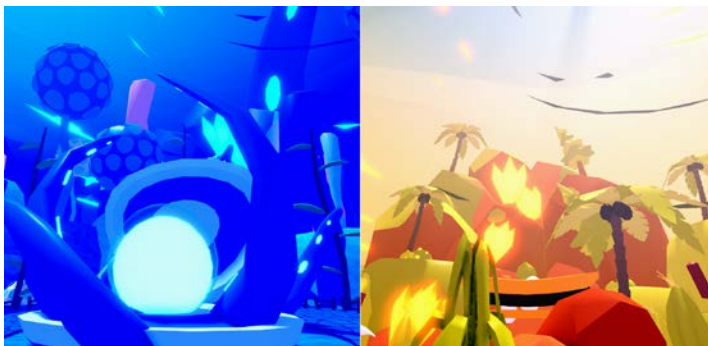
Figure 8.4: ProximityPrompt appearance can be highly customized

You will need to design the UI that will override the default appearance as well as use different properties and events of the ProximityPrompt to ensure correct behavior.

## Make a world lighting system

Something that can add a lot more ambience to your experience is a world lighting system. If you are making an experience with multiple separated locations, this can be especially effective. For example, the desert is likely to have a hot sun with blue skies while a winter wonderland is like to have a white sky and be hazy. By keeping track of what world the player is in, you can change lighting properties accordingly. In Figure 8.5, we can see two very different lighting themes taken from within the same experience.

Since players will almost always be in different worlds at the same time, this effect is local, meaning a player in a world with different lighting does not affect the lighting for a player in another:



*Figure 8.5: You can create much more immersive environments with a world lighting system*

## Make a projectile system

In *Chapter 6, Creating a Battle Royale Game*, we made a weapon system that used Raycasts to check hit detection. The only downside to this system is that hit detection was instantaneous, meaning there was no projectile moving from our gun to the target like in real life. With what you have learned, you should be able to make a system that simulates projectiles for hit detection. Keep in mind that the optimal solution would be to still use Raycasts for hit detection but to update where you are casting to and from until a target is hit. Think of what settings you would want for this type of system, such as projectile velocity and gravitational acceleration.

## Make a car system

Cars are part of many popular Roblox experiences as they're fun to drive, highly customizable, and can provide advantages depending on the experience's objective. You should make a car system to add to your own experiences. You should keep in mind that what you're really making is the chassis of the vehicle, that is the wheels, suspension, and other functional parts of the car. Your system should allow for different car body models to be added easily so that there is minimal reworking, aside from tuning, when adding new cars to your experience. Keep in mind what type of settings would be good to include for your car system, such as turning speed, acceleration, max speed, and more.

## Make a racing system

Racing is a simple competition that is present in a lot of Roblox experiences. Players can race on foot, with cars and other vehicles, pets, and more. You should create a racing system that supports tracking the positions of different race participants, multiple laps, and reports at the end the times as well as positions of different participants. Remember that some players might try to break your system by finding flaws in your map or simply won't finish. To fix the former issue, make sure to have multiple checkpoints throughout your racetrack and that your system makes sure players reach them. For the latter, enforce a time limit for the race to be finished within. If a player doesn't finish the race in time, you can simply say their time was DNF (Did Not Finish). By adding this system to your experience, you can add a new competitive element to your experience and give players new ways to earn rewards.

## Make an aircraft system

Depending on how accurate to real life you want to make your aircraft, this can be a rather advanced system to make. However, you can make simple aircraft that fly as most players would expect using the various movement constraints that were discussed in *Chapter 4, Roblox Programming Scenarios*. Think about what controls will make the most sense to players. For example, using keys only might be easier to program but controlling the aircraft by the direction of your mouse, controller, or mobile joystick is much nicer and probably what most players expect. Once again, consider what settings make sense for this type of system, roll speed, acceleration, top speed, max altitude, and more.

## Make a ship system

Ships are another type of vehicle you can create if your experience has water elements. There are many approaches to making a ship in Roblox, but the easiest approach is again to use movement constraints. Remember to add various settings so that making different types of ships is as easy as possible for you. Once you create this system, players will be able to sail the high seas in search of treasure, take leisurely cruises, and more!

## Make fighting NPCs

While we showed **Player versus Player (PvP)** examples in *Chapter 6, Creating a Battle Royale Game*, many experiences will choose to include **Player versus Environment (PvE)**, that is, fighting NPCs or AI. Enemy NPCs may take many shapes and sizes but for this example, imagine you want to make an experience where players work together to fight off zombies.

You will need to create these zombies to be aware of the closest player and move toward them, likely by way of `PathfindingService`. `PathfindingService` is a service that allows for the creation of paths from one point to another, avoiding obstacles or jumping over gaps when necessary. To learn more about how to use `PathfindingService`, visit the page here:

<https://developer.roblox.com/en-us/articles/Pathfinding>

Once the zombie is close enough, it should attack the player, dealing damage to them. Your system should control NPCs from one central location, like a module. To keep your system dynamic and make it easy to add new NPC types, perhaps put different attacks and behaviors in a table associated with that NPC type. With this system made, you can create a wide variety of different games where NPCs interact with players. Note that the NPCs don't always need to attack – with your system set up as described, you could add behaviors to emulate animals and more.

## **Make a survival system**

In real life, there are a few things we need to survive: food, water, shelter, and air to name a few. If you want to make your experience more realistic you can make it so that players also need these essential things. You can create a system that tracks the hunger, thirst, and air of a player. If the player eats or drinks, their hunger and thirst levels should both go down respectively. If they are underwater, their air level should go down and be replenished when they resurface. A system like this may have more limited applications outside of a survival game but is still good practice for making systems for other types of games.

## **Create an inventory system**

Having an inventory system in your experience is a great way to make players feel more physically connected with the virtual world around them. Depending on the genre of your experience, having a limited amount of space adds a sense of challenge and realism. For this system, you should create a UI menu that fills with items as you collect them from the physical world. You can collect items from the world with the interaction system you made in the *Create an interaction system* section of this chapter. If you want to make your system more advanced, perhaps allow for stacking of items of the same type. Stacking here means if you have three blocks of wood, they will take up one space and display that there is a quantity of three of that item instead of taking up three separate spaces. You could then enforce a stack limit through a setting that you implement. If a player's inventory is full, you should notify them of this if they try to collect more items. This system is a great addition to experiences in the genre of survival, horror, and even roleplay.

## Make a pet system

Pets are a very popular system in Roblox experiences. Pets, in the Roblox sense, can be actual animals or more abstract or fantastical creatures like unicorns or centaurs, but they typically serve to give the player a multiplier in their earnings or to automate the player's tasks such as automatically generating clicks or currency. In most experiences, players unlock pets by spending in-game currency at a dispenser that has pets of different rarities contained within it. You should make a pet system that allows for players to equip pets they've unlocked from an inventory and similarly unequip them. When equipped, the pets should follow the player around unless the pet is carrying out some task or other action. You can see an example of a pet following a player in *Figure 8.6*:



*Figure 8.6: Pets follow the player around and offer boosts such as multipliers*

When you unequip your pet, it can simply disappear, or you can add some other effect. Movement constraints will once again be one of the best ways to move the pet models around. Remember to think of edge cases, such as what happens to the pets if the player dies? You'll want smooth, predictable behavior, so the pets should simply move to where the player has respawned in this case.

## Make a crafting menu

A crafting system lets players use collected items or resources to create new items. Crafting is a feature in many experiences across several genres. Making some items only obtainable through crafting in your experience can be a great tool for pacing or adding more of a challenge. Your crafting menu should show what resources or items your player has available, what items can be crafted, and what resources are needed for those items to be crafted. If you want inspiration, try looking at some of the many survival games on Roblox or the crafting system from Minecraft.

## Create a house customization system

Customization is something Roblox players love, be it their Roblox characters, vehicles, or in-experience houses and properties. You should make a system where players can purchase and set down furniture and other décor on their property but not the property of others. You should break this system down into several parts. Firstly, the placement system should let players place items down where their mouse or other pointer is, assuming where they want to place it down is their property and doesn't collide with an item that has already been placed. You can see an example of this in *Figure 8.7*, where the furniture item is blue when it can be placed and red when one of the listed conditions isn't met:



*Figure 8.7: You can add visual indicators to show when placement is allowed*

Secondly, they should have an inventory system for the different items they can put down in their house. These items should most likely be gotten from some sort of store in your experience. When an item is placed down it should be removed from the player's inventory. Players should also be able to remove items; when they do, the item should go back into their inventory. Each system by itself isn't too difficult but tying everything together can be complicated. Remember if you're finding any part of this system difficult to split up each part further.

Now that we have explored different systems that can enhance any experience, we'll look at some genres of experiences you might think to create an experience after and what some of the necessary systems for those experiences might include.

## Experience ideas

In this section, we will explore what types of experiences you might think to create on the Roblox platform as well as the underlying systems you would need to make as a programmer. Keep in mind that new genres of experiences are always emerging, sometimes completely original or as the combination of several genres.

## Simulators

Simulator games are extremely popular on Roblox. In these experiences, players typically start off as “weak” or with poor equipment and through some simple tasks, such as clicking, they gain something they can exchange for currency or gain currency directly. They can then spend what they earn on better equipment that allows them to earn more money and progress faster. Many simulators have multiple worlds or maps so that players have a better sense of progression. For example, after unlocking the first 10 equipment items, you must unlock and move to the next world to purchase the next 10. Often, simulator games include pets so consider making a pet system that includes pets relevant to the theme of your experience. Next, we'll look at some, but not all, systems you would need to consider if you wanted to make a simulator game:

- **Player data system:** Like in previous examples, we will need to record what equipment the player has unlocked, their currency, what pets they have and whether they're currently equipped, and what worlds they've unlocked.
- **Pet system:** Pets follow your character around. You'll need to make a system, which is client-sided in most cases, that controls the movement of equipped pets around a player's character. Some of the movement constraints covered in *Chapter 4, Roblox Programming Scenarios*, may be helpful in accomplishing this.



- **Item shop:** This can be a UI-based or physical shop where players will buy their new equipment. For example, a simulator centered around weightlifting might sell better weights to lift in the shop that give you more strength or currency per click.
- **Premium shop:** If you want to maximize your earnings, you should include a UI-based shop for selling your passes and developer products. Remember to make what you sell attractive to potential buyers by offering a fair boost but don't ruin the experience for free players.

## Tycoons

Tycoons are a genre of Roblox experiences that have been popular for a long time. The objective of a tycoon game is that you have some plot of land and need to earn money to develop it or unlock different parts of it. For example, in a “mining tycoon” experience, the player might start with an empty plot of land and either earns some small amount of money passively or by actively clicking something like ore to earn money. When the player has enough money, they can buy a stationary mining tool that generates money automatically. This process is repeated, buying more stationary equipment as well as the structure of the mine, be those walls or decorations until they've unlocked everything available. Since there is a well-defined end to the game, players don't necessarily come back to play tycoons once they finish them. Additionally, given how saturated the genre is, tycoons have become less popular over time. Most recently, a twist on the genre is to be some sort of shop owner, be that a restaurant, salon, grocery store, or arcade, and NPCs enter and spend money. This is a small but exciting twist on the genre and most experiences allow players to arrange their tycoon space with different items as they wish instead of buying equipment in predetermined locations. This is much better as players can use their own imagination and design their business how they want. Next, we'll look at some of the systems that would need to be made if you wanted to create an experience in this genre:

- **Player data system:** As before, we will need to record what equipment the player has unlocked, their currency, and potentially more.
- **Item shop:** This only applies if you plan to follow the latter design mentioned where players can purchase different items and place them in their tycoon space themselves. This is best as a physical shop where players will be able to see and buy their new equipment, sort of like going to a hardware store. For example, if you were making a salon tycoon, you would want to sell styling chairs, decorations, and perhaps entirely different salon themes.

You can see an example of this in *Figure 8.8*:



*Figure 8.8: An example of a physical item shop in a tycoon experience*

- **Premium shop:** Like before, a UI-based shop will maximize your earnings. Remember to have a thought-out monetization strategy as discussed in the *Monetization* section of *Chapter 7, The Three Ms*.

## Roleplay games

Roleplay games are another popular genre of experiences on Roblox. One of their biggest advantages is that they have no end; most of the experience is simply social interaction and role playing with other users. One example of roleplay games is where there are adults and children, and parents can adopt the children and roleplay as a family. The possibilities for the content in the game are endless. Let's look at some of the systems you might need for this type of experience:

- **Player data system:** What data you're keeping track of can vary greatly depending on your experience. Players might earn money passively to buy different roleplay items, in which case you will need to track stats like currency.
- **Item shop:** This will also vary depending on what your experience is but it might make sense to include different roleplay items such as food or toys in your experience.
- **Premium shop:** Once again, you'll need to create a UI-based or physical shop for anything you want to sell for Robux in your experience.

## Hangout games

Hangout games can sometimes overlap with roleplay games, but the focus is more on real social interaction than roleplaying. For example, you could create a nightclub where people can sit on cozy-looking couches, listen to music, watch a lightshow, and more. You can see an example of this type of layout in *Figure 8.9*. These types of experiences are often heavily tied into the metaverse ideas discussed previously and can be great for making new friends or strengthening existing connections. Again, how you want to design an experience of this type can vary greatly; we'll look again at some systems you'll likely need:

- **Player data system:** Once again, the data you're keeping track of will be dependent on your experience. Make sure to keep track of things like currency and unlocked items.
- **Item shop:** This will again vary with the design of your experience, but it might make sense to include different roleplay items such as food or accessories in this genre of experience.
- **Premium shop:** A UI-based or physical shop should again be created for selling Passes or dev products.



*Figure 8.9: An example of hangout game design*

In the next section, we'll look at new Roblox features and some examples of how you can use them to make incredible enhanced experiences.

## Roblox features

This section will talk about some of the new features and tools Roblox has introduced since the last edition of this book.

## Group name changes

In January 2022, Roblox announced that developers would be getting the ability to change the name of a group. This had been a long-requested feature, and many expressed their appreciation for its release. If you own a Roblox group, you can change the name of it every 90 days for 100 Robux, assuming the name you wish to change it to is not already in use.

## Free badge creation

In February 2022, Roblox announced that developers would now be able to create badges for free, instead of the 100 Robux required before. In order to prevent abuse, developers can upload five badges in a 24-hour period for free and then any additional badges in that same time period will cost the original price of 100 Robux. As mentioned in *Chapter 5, Creating an Obby*, badges are another way to make players feel rewarded and further encourage competition among players of your experience.

## Spatial voice

Spatial voice (voice chat) was a highly anticipated new Roblox feature that was first released to developers in September 2021. With this feature, players who have access to spatial voice can talk to one another in any experience with the feature enabled. Unlike talking to someone on the phone, players speak from their character and how loud they are to you depends on how close they are, just like real life! Roblox being a social platform means that this feature can really improve interactions between players across different experiences, especially hangout games and similar experiences discussed. For most users, you need to confirm you are 13 years old or older by verifying your age with Roblox to get access to spatial voice. You can learn more about spatial voice and how to get access to it at the following page:

<https://devforum.roblox.com/t/spatial-voice-developer-beta/1479160>

## Mesh deformation

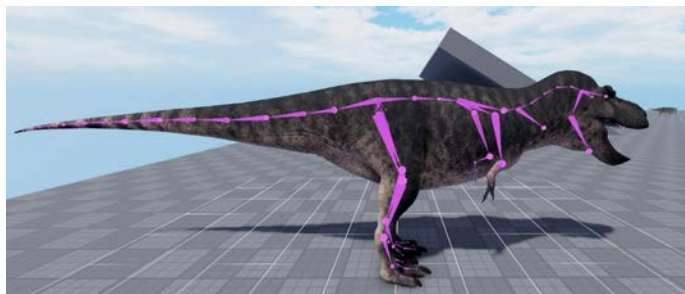
In late 2020, Roblox announced the introduction of skinned MeshParts. Previously, parts were rigid and Roblox characters, joints, movement, and other manipulation could only be represented by a separate part.

Skinned MeshParts allow for smooth and more realistic-looking behaviors. You can see an example of this in *Figure 8.10*:



*Figure 8.10: Mesh deformation allows for more realistic, polished character design*

Mesh deformation works through a series of “bones” around which the mesh forms itself. As these bones move, the mesh and textures move and bend with them. You can see an example of these bones as highlighted in *Figure 8.11*:



*Figure 8.11: Bones allow for mesh deformation and can be manipulated in Studio and other apps*

Mesh deformation can be a tricky concept to learn, especially at the beginning. If you're interested in learning how to use mesh deformation in your own experiences, you can visit the following article:

<https://devforum.roblox.com/t/skinned-meshparts-are-live/831011>

Since the first release of the technology, Roblox has added more features such as new underlying deformation techniques and expanded on its applications including layered clothing, which we will discuss next.

## Layered clothing

In October 2021, Roblox announced 3D layered clothing would become available for Roblox avatars. Layered clothing is a way to customize Roblox avatars with clothing that looks and moves more like real fabric than the simple textures used before. Layered clothing fits and forms over avatars and accessories because of additional geometry that is created during the modeling process. You can see examples of layered clothing in *Figure 8.12*:



*Figure 8.12: Some examples of layered clothing*

If you want to learn more about layered clothing and the special instances associated with this new technology, you can visit the following link:

<https://devforum.roblox.com/t/3d-layered-clothing-is-now-available/1517745>

## Flipbooks

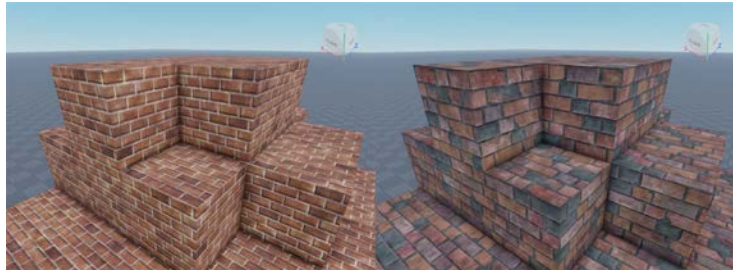
In March 2022, Roblox announced flipbooks. Flipbooks, or animated textures for particle emitters, allow you to create more detailed and animated particle effects. A flipbook is a sprite sheet, that is, a series of images or frames in a grid that are shown successively.

For example, if you wanted a single particle to show the beginning of an explosion before fading away to smoke, you can do so through this newly added feature. If you want to learn more about flipbooks, you can follow the link below:

<https://devforum.roblox.com/t/animated-particles-with-the-particle-flipbooks-beta/1718023>

## Custom materials

In December 2021, Roblox announced that they would be creating `MaterialService` for developers to create their own materials for terrain and `BaseParts`. This was a long-requested feature as many developers had felt that the Roblox-created materials simply did not always have the ability to convey their aesthetic decisions. You can see an example of legacy terrain materials and a user-created material in *Figure 8.13*:



*Figure 8.13: Custom materials can allow for different aesthetics than legacy Roblox materials*

If you want to learn more about `MaterialService` and its uses, visit the DevForum article linked here:

<https://devforum.roblox.com/t/upload-custom-materials-in-the-new-material-service-beta/1583158>

## Talent Hub

In August 2021, Roblox announced Talent Hub, a new resource where developers can hire other developers or advertise themselves to be hired. In the past, Roblox developers looking for work would often post their portfolios on the DevForum but there was not a dedicated section for this. As such, finding work or other developers qualified for your job was difficult and frustrating. With the creation of this site, you can quickly look for developers or find work by using filters that sort by development type, compensation type, job type, and more.

Additionally, developers can go through a process to be “verified,” which means Roblox reasonably thinks that the developer is a real person and is who they claim to be. You can find Talent Hub here:

<https://talent.roblox.com/>

## Other development types

Now, with the skills to make any program you want, problem-solving skills, and deeper knowledge about experience systems all under your belt, we will look at some of the cool things other development types do and that you may want to do yourself in addition to programming.

### Plugins

Plugins are powerful tools that can greatly increase your workflow in Roblox Studio. Making plugins is easy and you may decide to make some for your own use or to be used by others. Some developers choose to primarily make plugins and often sell them on the Creator Marketplace. Try creating one yourself! To learn more about creating plugins you can visit the following page:

<https://developer.roblox.com/en-us/articles/Intro-to-Plugins>

### UI/UX design

UI designers are those who create sets of UIs for experiences. Part of their work is making sure that the design of their UI is easy and clear to interact with; this is known as user experience design or UX. Additionally, they make sure that the UI they design is attractive to players and often their design will reflect the theme of the experience. For example, in a pirate-themed experience, pages of the UI set might include a helm, cannon, sail, and more. UI is an important part of your experience as players interact with it often.

### Art

There is a wide variety of developers on Roblox whose work can be considered art. In this section, we will explore some of those types of developer, what they do, and why they’re important.

### Clothing design

Traditionally, clothing designers on Roblox make texture-based clothing such as shirts, t-shirts, and pants for Roblox avatars. While there are many designers who still work primarily in this way, those who make 3D layered clothing or 3D character accessories also fall under this category. These designers work with their own imagination as well as adopting popular trends to create new items that let Roblox players express themselves across the metaverse via their avatars.



Thumbnails/icons

Some artists make their living creating either rendered or drawn art for thumbnails and icons of Roblox experiences. These designers use their talent to capture the mood and content of an experience and give an attractive first impression to potential players.

Particle design

Some developers focus specifically on designing images for use with particle emitters. These designers will often combine multiple particle effects along with meshes to make visually stunning effects for skills, world design, and more.

Sound and music design

Sound design is a very important aspect of making your players feel completely captivated by the scenes of your experience. Sound designers work to make sound effects to accommodate any sort of action, effect, or event in a scene. This can range from sci-fi explosions to calm natural ambiance.

Music designers also work with sound, making soundtracks that completely set the tone for an experience and correlate with what is taking place in a scene. Music designers use their knowledge of music as a science to develop engaging pieces to not just accompany but enhance experiences.

Animations

Maybe without realizing it, you’re looking at animations all the time. Your character uses an animation pack that was created by Roblox but animation artists are responsible for making animations seen across all types of experiences. As an animation artist, you can use Roblox or external software to bring different characters and objects to life in ways much too difficult to replicate with programming. While many animation artists choose to use external editing software such as Blender, you can make animation in Roblox Studio with user-made plugins such as Moon Animation Suite or by using the Roblox Animation Editor. You can find the Roblox Animation Editor plugin on the Plugins tab as seen in *Figure 8.14*:

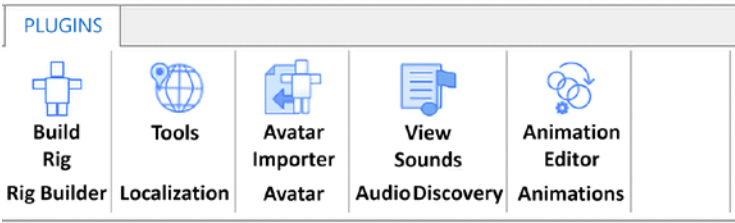


Figure 8.14: You can make and edit animations in Roblox Studio with third-party plugins or the Animation Editor

## Summary

In this chapter, you learned 50 cool things to do on Roblox and hopefully gained inspiration for many new projects through making different systems.

Now, we find ourselves here, at the end. But in reality, this is not the end. For you, you have arrived at the bright beginning of a journey of making lifelong friends, memorable game titles, professional connections, and money to support yourself in the future.

I wish you the best of luck in your endeavors!



# Worksheet Answers

## Chapter 1

Q1. What is a common and interchangeable word for games used on Roblox?

A: Experience.

Q2. What is the name of the process through which developers exchange Robux for real-world currency?

A: Developer Exchange (DevEx).

Q3. What do developers sell for Robux in their games in order to earn money? Specifically, what are used for one-time purchases and what are used for multiple purchases?

A: Passes are used for one-time purchases and developer products are used for multiple.

Q4. How long may it take for developers to receive the proceeds of in-experience sales?

A: It can take three to seven days.

Q5. What is the name of the programming language used in Roblox development?

A: Luau.

Q6. What do programmers do on Roblox?

A: Programmers create the core of any game experience, making any functionality that doesn't happen by default.

Q7. What do 3D modelers do on Roblox?

A: 3D modelers create the individual items in an experience including furniture, pets, food items, or any other visual pieces that are typically small to medium in size.

Q8. What do builders do on Roblox?

A: Builders fulfill the role of creating the worlds that exist in your experiences.

Q9. What do UI/UX designers do on Roblox?

A: UI/UX designers create the pages and screens that players interact with inside your game experiences.

Q10: What should you expect from your first projects on Roblox?

A: That they will be a learning experience.

## Chapter 2

Q1. What page should you visit if you want to start developing?

A: The **Create** page.

Q2. If you want to make your experience public or private, which menu should you visit?

A: The **Configure Experience** menu.

Q3. If you want to change the appearance of your game, such as the icon, thumbnail, or description, which menu should you visit?

A: The **Configure Place** menu.

Q4. If you want to change what platforms players can play your game on, or how many players can fit in a server, which menu should you visit?

A: **Configure Place > Access**.

Q5. When your game is ready to release and you want to promote it, which menu should you go to?

A: The **Sponsor This Experience** or **Advertise** menu.

Q6. If you want your work in Roblox Studio to be playable from the Roblox website, what must you do?

A: Publish your game via the **File** menu or by pressing *Alt + P*.

Q7. What is the word that refers to any kind of object in Roblox?

A: Instance.

Q8. Roblox uses what to change the behaviors and appearances of different instances?

A: Properties.

Q9. What are the five tools to physically manipulate objects in Roblox Studio?

A: The **Select**, **Move**, **Scale**, **Rotate**, and **Transform** tools.

Q10. Where can you quickly change many game-related settings from Roblox Studio?

A: The **Game Settings** menu.

Q11. How can you load in and test your experience in Roblox Studio?

A: By pressing the **Play** button under the **Test** tab.

Q12. What is the name of the forum where you can participate in developer discussions and get help and feedback from the Roblox developer community?

A: The Developer Forum (Dev Forum).

## Chapter 3

Q1. What is used in programming to hold a piece of data?

A: A variable.

Q2: What simple function can be used to send information to the **Output** window?

A: `print()`.

Q3: What data type should you use if you want to represent a value of true or false?

A: A Boolean.

Q4: What data type is used if you want to represent a word or characters?

A: A string.

Q5: What is used in programming when we want to check if a condition is met?

A: A conditional statement or `if` statement.

Q6: What is used in programming when some action must be done repeatedly so long as a condition is met?

A: A loop.

Q7: What loop would be best to use if we want something to happen an exact number of times?

A: A for loop, more specifically a numeric for loop.

Q8: What type of loop is best if we want something to always happen as long as some condition is met?

A: A while loop.

Q9: What type of loop is best if we want to do something immediately once and then loop only while some condition is met?

A: A repeat loop.

Q10: What do we call a repeatedly callable block of code that is typically designed to accomplish some single task or part of a task?

A: A function.

Q11: What is the part of a function called where the function's name and parameters are present?

A: The header.

Q12: What type of function can take any number of arguments without them being explicitly defined in the function's header?

A: A variadic function.

Q13: What is a function said to be if it calls itself?

A: Recursive.



Q14: What is a part of an instance that we say is “fired” when something happens?

A: An event.

Q15: What is a part of an instance that is callable and causes some behavior?

A: A method.

Q16: Why should you use good programming style no matter what?

A: There are many correct answers but mostly readability (by yourself or others), correct functionality, and potentially better performance.

## Chapter 4

Q1. What is the role of the client and the server in the client-server model?

A: The client acts as a requester and the server is a provider and verifier.

Q2. What three script types are primarily used in Roblox development?

A: Scripts (server-side), LocalScripts, and ModuleScripts.

Q3. What can you add at any line in your script to stop the program and examine its state?

A: A breakpoint.

Q4. What does `FilteringEnabled` do?

A: It restricts clients from directly making changes to the experience that replicate to other clients; instead clients must make requests to the server via `RemoteEvent` and `RemoteFunction`.

Q5. What instances are used for communication between different script instances on the same side of the client-server model?

A: A `BindableEvent` and `BindableFunction`

Q6. What high-level instances are used for organizing and adding more complex behaviors, additional events, and methods?

A: `Services`.

Q7. What services are best for storing assets?

A: `ServerStorage` and `ReplicatedStorage`; remember that clients do not have access to assets stored in `ServerStorage`.

Q8. What service is used for detecting physical input from the player?

A: `UserService`.

Q9. What service can have its properties modified to change default character behaviors?

A: `StarterPlayer`. Note that these properties can also be changed via the **Game Settings** menu.

Q10. What base class of instances is often used for physics manipulation of `BasePart`?

A: `Constraints`.

## Chapter 5

Q1. What is the purpose of structuring your experience to be modular?

A: Keeping systems in separate modules is great for organization and lets them run independently of each other.

Q2. What service is used for storing and retrieving player data?

A: `DataStoreService`.

Q3. What is the purpose of including a `recursiveCopy` function in the `Data` module?

A: Tables are assigned to a variable by a shared reference instead of the structure being copied for performance. We want a unique table at several points throughout this system.

Q4. Why do we wrap asynchronous methods in a `pcall()` function?

A: These methods can produce errors; we need to make sure these errors don't crash our program and retry when necessary.

Q5. What is the importance of including the `ProcessReceipt` event function in your game when selling developer products?

A: Roblox uses this to verify sales of products; you will not receive your earnings without it.

Q6. Why might it be important to spend time adding sounds, particles, and other effects into your experience?

A: Players enjoy this polish and associate them with reward, potentially improving engagement.

Q7. What method of a `RemoteEvent` can be fired so that the server can communicate with an individual client?

A: `FireClient()`.

Q8. Why, when using `RunService` loops, might it be important to use the passed `dt` (short for `delta`) value?

A: This value represents the time between the current and last frame/loop. Depending on your effect, failing to factor in this value can cause different effect speeds for those with different frame rates.

Q9. What key can you press while in your experience or play testing in Roblox Studio to see errors from the client or server?

A: Pressing *F9* will bring up the **Developer Console** where you can see errors and much more.

## Chapter 6

Q1. Why is it good to keep weapon settings in a module?

A: Keeping settings in a module means you won't need to interact with weapon system code when making changes.

Q2. Why is a lobby necessary in experiences of this format?

A: Players need somewhere to be when they're eliminated. They will either spectate or interact with shops or other attractions in the lobby in the meantime.

Q3. What do you need to have in your map for every player that can be in your experience at once?

A: A spawn location part.

Q4. Why might it be good to include a celebration message or effect showing the victor at the end of a round?

A: It makes the winner feel more rewarded and encourages more competition among players.

Q5. Why is local replication important for different effects?

A: While we could create effects on the server, latency, lower calculation refresh rates, and more result in the effects being less smooth. Replicating to each client makes things smooth and predictable.

Q6. Where can you find many quality weapons models in addition to millions of other assets?

A: The **Toolbox**.

Q7. What can you add to rare weapons and items to make finding them more exciting?

A: Particles, sounds, trails, and other effects.

Q8. Why might a UI-based shop be better than a physical shop?

A: A UI-based shop is always present and easier to navigate than physical shop parts. As such, you may make more revenue by including one.

## Chapter 7

Q1. What are the Three Ms?

A: Mechanics, Monetization, and Marketing.

Q2. What is the basic game concept of a simulator?

A: Players start off basic or with poor tools, weapons, or abilities. Over time, after performing a simple task they can upgrade and become more powerful.

Q3. What is one of the greatest aspects of RP games in terms of engagement?

A: There is no defined end to the game, so users will often return for the gameplay, which is mostly social interactions.

Q4. What is something you should always avoid when implementing your monetization strategy?

A: Making your experience “pay-to-win,” that is, locking high-tier items behind a simple paywall.

Q5. What is a good level of advantage to be expected from Passes?

A: Mid- to upper-mid-tier range items, where the Pass serves more as a progression aid than an overall gameplay advantage.

Q6. What is a truly fair monetization option that may bring in less overall revenue?

A: Selling only cosmetic additions.

Q7. In what two ways can you monetize access to your experience?

A: You can sell access to your experience or sell a private server subscription.

Q8. What two types of experience promotion formats does Roblox offer?

A: Users Ads and Sponsored Experiences.

Q9. What is a grouping of experiences by genre, engagement, or other factors called on Roblox?

A: A sort.

Q10. What external group of content creators can be crucial in getting your experience publicized?

A: YouTubers.

Q11. What is one method to maximize potential performance of future experiences?

A: Develop a community around your work and promote upcoming titles ahead of release via Roblox groups, Discord, and other platforms.

Q12. What trend does Roblox have the potential to lead and change the way humans interact?

A: The Metaverse.

## Invitation to join us on Discord!

Hope you have already joined us on Discord. It is a great place to network with Roblox users and to keep yourself up to date with Roblox news, latest features and most importantly to up-skill with peers and experts around you.

Do check-out the #resources channel where Zander shares links to interesting news on Roblox and resources to help you get started.

Scan the QR code or visit the link to join the community.



<https://packt.link/letscreategames>



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

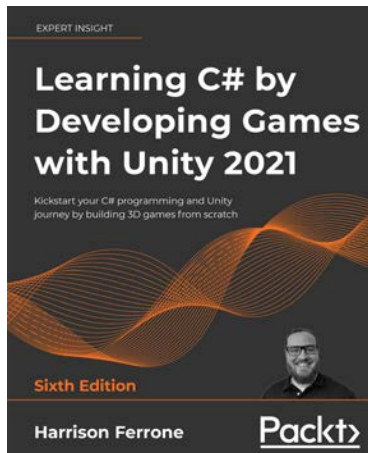
At [www.packt.com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.





# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



## **Learning C# by Developing Games with Unity 2021, Sixth Edition**

Harrison Ferrone

ISBN: 9781801813945

- Follow simple steps and examples to create and implement C# scripts in Unity
- Develop a 3D mindset to build games that come to life
- Create basic game mechanics such as player controllers and shooting projectiles using C#
- Divide your code into pluggable building blocks using interfaces, abstract classes, and class extensions

- Become familiar with stacks, queues, exceptions, error handling, and other core C# concepts
- Learn how to handle text, XML, and JSON data to save and load your game data
- Explore the basics of AI for games and implement them to control enemy behavior



## 101 UX Principles, Second Edition

Will Grant

ISBN: 9781803234885

- Work with user expectations, not against them
- Make interactive elements obvious and discoverable
- Optimize your interface for mobile
- Streamline creating and entering passwords
- Use animation with care in user interfaces
- How to handle destructive user actions

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Share your thoughts

Now you've finished *Coding Roblox Games Made Easy, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

# Index

## Symbols

3D modelers 7

tables

reversing 231

## A

aircraft system

creating 239

announcement system

creating 234, 235

Application Programming

Interfaces (APIs) 39, 219

argument 74

arrays 49

art 251

assert() function 75

Avatar Shop 23, 24

## B

backend

setting up 122, 123, 166

base classes 66

Battle Royale Game

backend, setting up 166

frontend, setting up 196

local replication 189-192

loot, spawning 192-196

player data, managing 166-168

round system, setting up 181-184

weapons, creating 168-181

working, with UI 196

**Battle Royale Game, round system**

player, preparing 184-189

**Battle Royale Game, UI**

game message 197-201

players display 197-201

shop, creating 207, 208

spectate menu, making 201-206

**Beams 117**

**BindableEvents 95, 96**

**BindableFunctions 95, 96**

**Boolean data type**

setting 53

**booleans 49**

**bools 49**

**builders 7**

## C

**callback 94**

**car system**

- creating 238
- CFrame** 50, 61-63
- classes 50
- Click-Through Rate (CTR)** 23
- client-server model** 86
  - BindableEvents 95, 96
  - BindableFunctions 95, 96
  - FilteringEnabled 91
  - local scripts 87
  - modules 87-89
  - RemoteEvents 92, 93
  - RemoteFunctions 95
  - Script Menu tab 89-91
  - scripts 86, 87
  - script types 86
- collision groups** 101
- community**
  - creating 223
- compound operators** 52
- conditional expressions** 64-68
- conditional statements** 64
- Configure Experience menu** 15, 16
- Configure Localization option** 20
- Constraint** 104-107
- crafting menu**
  - creating 242
- Create Badges option** 20-23
- Create page**
  - traversing 12-14
- Creator Marketplace** 23, 24
- custom materials** 250
- custom ProximityPrompt appearance**
  - creating 236, 237

## D

- daily reward system**
  - creating 235
- data types** 48-50
- debounce** 138
- Developer Exchange** 4
  - reference link 5
- Developer Forum** 42
  - reference link 42
- DevEx** 4
- dev products** 4
- dictionaries** 49, 57-59

## E

- Electronic Arts Inc. (EA)** 218
- events** 73, 80
- Experience setting**
  - configuring 14
- Explorer**
  - features 32
  - utilizing 29-31

## F

- Fibonacci sequence**
  - nth number, searching of 233
- fighting NPCs**
  - creating 239
- FilteringEnabled** 91
- FizzBuzz** 230
- flipbooks** 249
- floating-point errors** 48

**for loops 68-71**

- generic for 68
- numeric for 68

**free badge**

- creating 247

**frontend**

- setting up 155, 196

**functions 73**

- in programming 73

**G****game**

- guessing 232, 233

**game development, peripheral experience aspects**

- effects 117, 118
- lighting 114-117
- sound 111-114

**games**

- roleplay 245

**Graphical user interfaces (GUIs) 99, 181****group name**

- modifying 247

**H****hangout game design 246****header 73****house customization system**

- creating 242, 243

**I****in-experience currency shops**

- creating 152-154

**in-experience purchases**

- types 4

**instances 50, 64**

- events 79, 80
- methods 79-81

**interaction system**

- creating 235

**inventory system**

- creating 240

**items 70****iterator function 69****L****layered clothing 249****leaderboard system**

- creating 233, 234

**linear equation**

- solving 232

**local replication 189-192****local scripts 87****logical operators 65****loops**

- declaring 68
- for loops 68-71
- repeat loops 73
- using 68
- while loops 71, 72

**loot**

- spawning 192-196

**Luau 7****M****marketing 220**

- advantages and disadvantages 221
- community 223
- Roblox promotion system 220
- YouTubers 222



**mechanics 214**

- minigames 218
- RP games 216
- simulators 214, 215
- tycoons 216, 217

**mesh deformation 247-249****minigames 218****modules 87-89****monetization 218, 219****movement constraints 107-111****Move tool 33****multithreading 76****N****naming convention 82****nesting 58****non-player characters (NPCs) 58, 180, 217****nth number**

- searching, of Fibonacci sequence 233

**number of stickers 231****number variables 52****O****obby stages**

- creating 134
- exploits, preventing 155
- in-experience currency shops,  
creating 152-154
- part behaviors, creating 135-142
- rewards, creating 142, 143
- Robux premium purchases 145-151
- shops and purchases 144

**obstacle course game (obby) 102, 121**

- backend, setting up 122, 123
- frontend, setting up 155

- testing and publication 161-163

**obstacle course game (obby), backend**

- collisions and player characters,  
managing 132, 134
- player data, managing 123

**obstacle course game (obby) backend,  
player data**

- datastore system, creating 123-125
- edge case, addressing 130-132
- saving 129, 130
- session data, creating 125-127
- session data, loading 125-127
- session data, manipulating 127, 128
- throttling, addressing 130-132

**obstacle course game (obby), frontend**

- effects, creating 156

**obstacle course game (obby) frontend,  
effects**

- particles 157
- part movement 159-161
- sound 156, 157
- tying 158

**OrderedDataStores**

- reference link 234

**P****parameter 74****ParticleEmitters 117****parties**

- using 218

**passed-in number x**

- divisible, by passed-in number y 230

**peripheral experience aspects**

- adding 111

**pet system**

- creating 241

**physics**

- Constraint 104-107
- movement constraints 107-111
- working with 104

**PhysicsService 101, 102****place options**

- Configure Localization option 20
- Create Badge 23
- Create Badge option 20, 21

**Place settings**

- configuring 14

**player**

- data, managing 123-125, 166-168
- preparing 184-189

**Players service 96-98****Player versus Environment (PvE) 239****Player versus Player (PvP) 239****plugins 251****PolicyService 101****private servers 19****Private setting**

- versus Public setting 15

**programmers 7****programming**

- functions 74-76

**programming, style and efficiency**

- demonstrating 81
- general style rules 81, 82
- Roblox-specific rules 82

**projectile system**

- creating 238

**protected call (pcall()) 130****ProximityPrompts**

- reference link 235

**R****racing system**

- creating 239

**readability 81****recursion 76-79****relational operators 65****RemoteEvents 92, 93****RemoteFunctions 95****repeat loops 73****ReplicatedStorage 98, 99****Restitution 106****Roblox**

- clothing design 251
- Create page, traversing 12-14
- Developer Forum 42
- development types 251
- experience 12
- experience systems 233
- resources, advantages obtaining for 41
- Sound and music design 252
- Talent Hub 42
- thumbnails and icons 252
- tutorials and resources 41

**Roblox, Create page**

- Avatar Shop 23, 24
- Configure Experience menu 15, 16
- Creator Marketplace 23, 24
- Experience setting, configuring 14
- Place options 19
- Place setting, configuring 14
- Start Place menu, configuring 16

**Roblox developer**

- 3D modelers 7
- builders 7
- perspective, obtaining 8, 9

- programmers 7
- types, discovering 6
- UI/UX designers 7

**Roblox development**

- benefits 4
- financial opportunities 4, 5
- networking, benefits 5, 6
- professional skills, improving 5

**Roblox, development types**

- animations 252
- art 251
- plugins 251
- UI Design 251
- UX Design 251

**Roblox, experience systems**

- aircraft system, creating 239
- announcement system, creating 234, 235
- car system, creating 238
- crafting menu, creating 242
- custom ProximityPrompt appearance, creating 236, 237
- daily reward system, creating 235
- fighting NPCs, creating 239
- house customization system, creating 242, 243
- interaction system, creating 235
- inventory system, creating 240
- leaderboard system, creating 233, 234
- pet system, creating 241
- projectile system, creating 238
- racing system, creating 239
- ship system, creating 239
- survival system, creating 240
- world lighting system, creating 237, 238

**Roblox features 246**

- custom materials 250
- flipbooks 249

- free badge creation 247
- group name changes 247
- layered clothing 249
- mesh deformation 247-249
- programming challenges 229
- spatial voice 247
- Talent Hub 250

**Roblox features, programming challenges**

- element existence, checking in table 230
- FizzBuzz 230
- game, guessing 232, 233
- linear equation, solving 232
- maximum value, searching in table 230
- nth number, searching of Fibonacci sequence 233
- number of stickers 231
- passed-in number x, divisible by passed-in number y 230
- seconds, formatting into hours:minutes:seconds 230
- table, reversing 231
- table, sorting with sorting algorithm 232
- table, sorting with table.sort() 231, 232
- two tables, concatenate 231
- unique elements, returning from table 231

**Roblox platform, experience ideas**

- games, roleplay 245
- hangout games 246
- simulators 243
- tycoons 244, 245

**Roblox promotion system 220, 221****Roblox, services**

- PhysicsService 101, 102
- Players service 96-98
- PolicyService 101
- ReplicatedStorage 98, 99
- ServerStorage 98, 99

- StarterGui 99
- StarterPack 99-101
- StarterPlayer 99-101
- UserInputService 102, 103
  - using 96

## **Roblox-specific rules 82**

### **Roblox Studio**

- camera manipulation 28, 29
- customizing, to aid workflow 40, 41
- Explorer, utilizing 29-32
- File menu 26-28
- Game Settings menu, managing 34, 35
- movement manipulation 28, 29
- settings 26-28
- Test tab 37-39
- tools, using 32
- View tab 36, 37
- working with 25

### **Roblox Studio, tools**

- Move tool 33
- Rotate tool 33
- Scale tool 33
- Select tool 32
- Transform tool 34

### **Roblox Twitter Community (RTC)**

- reference link 6

### **Roblox YouTubers 6**

### **Robux 4**

### **Robux premium purchases 145-151**

### **Rod constraint 106**

### **role-playing games (RPGs) 216**

### **Rope constraint 106**

### **Rotate tool 33**

### **round system**

- setting up 181-184

## **S**

### **sanity checks 93**

### **Scale tool 33**

### **scope 70**

### **script**

- types 86

### **script injectors 87**

### **Script Menu tab 89-91**

### **seconds**

- formatting, into hours:minutes:seconds 230

### **Select tool 32**

### **ServerStorage 98, 99**

### **ship system**

- creating 239

### **short-circuit logic 68**

### **simulators 214, 215, 243**

### **Slack 6**

### **sorting algorithm**

- used, for sorting table 232

### **spatial voice 247**

### **Sponsored Experiences 21**

### **Sponsors 220**

### **StarterGui 99**

### **StarterPack 99-101**

### **StarterPlayer 99-101**

### **Start Place menu**

- Access tab 17-19
- configuring 16
- icon 17

### **store variable 124**

### **string library functions 55**

### **strings 49, 53-55**

**survival system**

creating 240

**T****table**

element existence, checking 230  
maximum value, searching in 230  
sorting, with sorting algorithm 232  
unique elements, returning from 231

**tables 55-57**

sorting, with `table.sort()` 231, 232

**table.sort()**

used, for sorting table 231, 232

**Talent Hub 42, 250**

reference link 42

**ternary expression 67****throttling 130****Trails 118****Transform tool 34****two tables**

concatenating 231

**tycoons 216, 217, 244, 245****U****UI Design 251****UI/UX designers 7****User Ads 21, 220****User-Generated Content (UGC) 24****UserService 102, 103****user interface (UI) 221**

working with 196

**UX Design 251****V****variables**

creating 48  
manipulating 51  
setting 51

**variadic function 75****vector 172****vectors 59, 60****VIP servers 19****W****weapons**

creating 168-181

**while loops 71, 72****world lighting system**

creating 237, 238

**Y****YouTubers 222**



